CANNING.001CP2

PATENT

SPARSE AND EFFICIENT BLOCK FACTORIZATION FOR INTERACTION DATA

Reference to Related Applications

The present application is a continuation-in-part of U.S. Patent Application No. 10/354,241, filed January 29, 2003, titled "COMPRESSION OF INTERACTION DATA USING DIRECTIONAL SOURCES AND/OR TESTERS," which is a continuation-in-part of U.S. Patent Application No. 09/676,727, filed September 29, 2000, titled "COMPRESSION AND COMPRESSED INVERSION OF INTERACTION DATA," the entire contents of both prior applications are hereby incorporated by reference. The present application also claims priority from U.S. Provisional Application No. 60/175,454, filed January 10, 2000, the entire contents of which is hereby incorporated by reference. The present application also claims priority benefit of U.S. Provisional Application No. 60/396,133, filed July15, 2002, titled "SPARSE AND EFFICIENT BLOCK FACTORIZATION FOR INTERACTION DATA," the entire contents of which is hereby incorporated by reference.

Computer Program Listing

A computer program listing in Appendix A lists a sample computer program for one embodiment of the invention.

Background of the Invention

Field of the Invention

The invention relates to methods for compressing the stored data, and methods for manipulating the compressed data, in numerical solutions such as, for example, antenna radiation-type problems solved using the method of moments, and similar problems involving mutual interactions that approach an asymptotic form for large distances.

Description of the Related Art

Many numerical techniques are based on a "divide and conquer" strategy wherein a complex structure or a complex problem is broken up into a number of smaller, more easily solved problems. Such strategies are particularly useful for solving integral equation problems involving radiation, heat transfer, scattering, mechanical stress, vibration, and the like. In a typical solution, a larger structure is broken up into a number of smaller structures, called elements, and the coupling or interaction between each element and every other element is calculated. For example, if a structure is broken up into 16 elements, then the inter-element mutual interaction (or coupling) between each element and every other element can be expressed as a 16 by 16 interaction matrix.

As computers become more powerful, such element-based numerical techniques are becoming increasingly important. However, when it is necessary to simultaneously keep track of many, or all, mutual interactions, the number of such interactions grows very quickly. The size of the interaction matrix often becomes so large that data compression schemes are desirable or even essential. Also, the number of computer operations necessary to process the data stored in the interaction matrix can become excessive. The speed of the compression scheme is also important, especially if the data in the interaction matrix has to be decompressed before it can be used.

Typically, especially with radiation-type problems involving sound, vibration, stress, temperature, electromagnetic radiation, and the like, elements that are physically close to one another produce strong interactions. Elements that are relatively far apart (usually where distance is expressed in terms of a size, wavelength, or other similar metric) will usually couple less strongly. For example, when describing the sound emanating from a loudspeaker, the sound will change in character relatively quickly in the vicinity of that speaker. If a person standing very near the speaker moves one foot closer, the sound may get noticeably louder. However, if that person is sitting at the other end of a room, and moves one foot closer, then the change in volume of the sound will be relatively small. This is an example of a general property of many physical systems. Often, in describing the interaction of two nearby objects, relatively more detail is needed for an accurate description, while relatively less detail is needed when the two objects are further apart.

As another example, consider a speaker producing sound inside a room. To determine the sound intensity throughout that room, one can calculate the movement (vibration) of the walls and objects in the room. Typically such calculation will involve choosing a large number of evenly spaced locations in the room, and determining how each location vibrates. The vibration at any one location will be a source of sound, which will typically react with every other location in the room. The number of such interactions would be very large and the associated storage needed to describe such interactions can become prohibitively large. Moreover, the computational effort needed to solve the matrix of interactions can become prohibitive. This computational effort depends both on the number of computations that must be performed and on the speed at which these computations are executed, such as on a digital computer.

Summary of the Invention

The present invention solves these and other problems by providing a compression scheme for interaction data and an efficient method for processing the compressed data without the need to first decompress the data. In other words, the data can be numerically manipulated in its compressed state. This invention also pertains to methods for processing the data with relatively fewer operations and methods for allowing a relatively large number of those operations to be executed per second.

Given a first region containing sources relatively near to each other, and a second region containing sources relatively near to each other, but removed from the first region; one embodiment provides a simplified description of the possible interactions between these two regions. That is, the first region can contain a relatively large number of sources and a relatively large amount of data to describe mutual interactions between sources within the first region. In one embodiment, a reduced amount of information about the sources in the first region is sufficient to describe how the first region interacts with the second region. One embodiment includes a way to find these reduced interactions with relatively less computational effort than in the prior art.

For example, one embodiment includes a first region of sources in one part of a problem space, and a second region of sources in a portion of the problem space that is removed from the first region. Original sources in the first region are modeled as composite sources (with relatively fewer composite sources than original sources). In one embodiment, the composite sources are described by linear combinations of the original sources. The composite sources are reacted with composite testers to compute interactions between the composite sources and composite testers in the two regions. The use of composite sources and composite testers allows reactions in the room (between regions that are removed from each other) to be described using fewer matrix elements than if the reactions were described using the original sources and testers. While an interaction matrix based on the original sources and testers is typically not a sparse matrix, the interaction matrix based on the composite sources and testers is typically a sparse matrix having a block structure.

One embodiment is compatible with computer programs that store large arrays of mutual interaction data. This is useful since it can be readily used in connection with existing computer programs. In one embodiment, the reduced features found for a first interaction group are sufficient to calculate interactions with a second interaction group or with several interaction groups. In one embodiment, the reduced features for the first group are sufficient for use in evaluating interactions with other interaction groups some distance away from the first group. This permits the processing of interaction data more quickly even while the data remains in a compressed format. The ability to perform numerical operations using compressed data allows fast processing of data using multilevel and recursive methods, as well as using single-level methods.

Interaction data, especially compressed interaction data and including data compressed by methods described herein, has a sparseness structure. That is, the data is often sparse in that many matrix elements are either negligible or so small that they may be approximated by zero with an acceptable accuracy. Also, there is a structure or pattern to where the negligible elements occur.

This sparseness structure can also occur in data from a variety of sources, in addition to from interaction data. For example, a number of computers that are connected by a network and exchange information over the network. However, the amount of data necessary to describe the complete state of each computer is much greater than the amount of data passed over the network. Thus, the complete set of data naturally partitions itself into data that is local to some computer and data that moves over the network. On each computer, the data can be ordered to first describe the data on that computer that is transmitted (or received) on the network, and then to describe the data on that computer that does not travel on the network. Alternatively, the data can be ordered to first describe the data that is shared among the computers, and second to describe the data that is not shared among the computers or is shared among a relatively small number of computers. A similar situation occurs with ships that communicate information amongst themselves, where a greater amount of information is necessary to describe the compete state of the ships.

A sparseness structure can include blocks that are arranged into columns of blocks and rows of blocks. Within each block there generally are nonzero elements. This data can be represented as a matrix, and in many mathematical solution systems, the matrix is inverted (either explicitly, or implicitly in solving a system of equations). Solution of the matrix equation can be done with a high efficiency by using a block factorization. For example, an LU factorization can be applied to the blocks rather than to the elements of a matrix. For some sparseness structures, this can result in an especially sparse factored form. For example, the non-zero elements often tend to occur in a given portion (for example, in the top left corner or another corner) of the blocks. The sparseness of the factored form can be further enhanced by further modifications to the factorization algorithm. For example, one step in the standard LU decomposition involves dividing by diagonal elements (which are called pivots). In one embodiment, sparseness results from only storing the result of that division for a short time. In one embodiment, it is possible to store the blocks where this division has not been done. These blocks often have more sparseness than the blocks produced after division.

A block factorization of interaction data has other advantages as well. By storing fewer numbers, fewer operations are needed in the computation. In addition, it is possible to perform these operations at a faster rate on many computers. One method that achieves this faster rate uses the fact that the non-zero elements can form sub-blocks of the blocks. Highly optimized software is available which multiplies matrices, and this can be applied to the sub blocks. For example, fast versions of Basic Linear Algebra Subroutines (BLAS) can be used. One example of such software is the Automatically Tuned Linear Algebra Subroutines (ATLAS). The use of this readily available software can allow the factorization to run at a relatively high rate (many operations executed per second).

Brief Description of the Figures

The advantages and features of the disclosed invention will readily be appreciated by persons skilled in the art from the following detailed description when read in conjunction with the drawings listed below.

Figure 1A illustrates a wire or rod having a physical property (e.g., a current, a temperature, a vibration, stress, etc.) $I(\ell)$ along its length, where the shape of $I(\ell)$ is unknown.

Figure 1B illustrates the wire from Figure 1A, broken up into four segments, where the function $I(\ell)$ has been approximated by three known basis functions $f_i(\ell)$, and where each basis function is multiplied by an unknown constant I_i.

Figure 1C illustrates a piecewise linear approximation to the function $I(\ell)$ after the constants I_i have been determined.

Figure 2 is a flowchart showing the process steps used to generate a compressed (block sparse) interaction matrix.

Figure 3 illustrates partitioning a body into regions.

Figure 4 shows an example of an interaction matrix (before transformation) for a body partitioned into five differently sized regions.

Figure 5 shows an example of an interaction matrix after transformation (but before reordering) for a body partitioned into five regions of uniform size showing that in many cases each group of non-zero elements tends to occupy the top left corner of a block.

Figure 6 shows an example of an interaction matrix after transformation and reordering for a body partitioned into five regions of uniform size.

Figure 7 illustrates the block diagonal matrix D^R.

Figure 8 is a plot showing the digits of accuracy obtained after truncating the basis functions for a block of the entire interaction matrix, with a block size of 67 by 93.

Figure 9 is a plot showing the digits of accuracy obtained after truncating the basis functions for a block of the entire interaction matrix, with a block size of 483 by 487.

Figure 10, consisting of Figures 10A and 10B, is a flowchart showing the process of generating a compressed (block sparse) impedance matrix in connection with a conventional moment-method computer program.

Figure 11 is a three-dimensional plot showing magnitudes of the entries in a 67 by 93 element block of the interaction matrix (before transformation) for a wire grid model using the method of moments.

Figure 12 is a three-dimensional plot showing magnitudes of the entries of the interaction matrix from Figure 11 after transformation.

Figure 13 shows an idealized view of a sparseness pattern for the intermediate results within the computation of a block of the factorization.

Figure 14 is a graph showing the time needed to compute the factorization of a matrix by various methods, where plusses show results for several problems solved by operating on sub-blocks.

In the drawings, the first digit of any three-digit number generally indicates the number of the figure in which the element first appears. Where four-digit reference numbers are used, the first two digits indicate the figure number.

Detailed Description

Many physical phenomena involve sources that generate a disturbance, such as an electromagnetic field, electromagnetic wave, a sound wave, vibration, a static field (e.g., electrostatic field, magnetostatic field, gravity field, etc) and the like. Examples of sources include a moving object (such as a loudspeaker that excites sound waves in air) and an electrical current (that excites electric and magnetic fields), etc. For example, the electric currents moving on an antenna produce electromagnetic waves. Many sources produce disturbances both near the source and at a distance from the source.

Sometimes it is convenient to consider disturbances as being created by an equivalent source (e.g., a fictitious source) rather than a real physical source. For example, in most regions of space (a volume of matter for example) there are a large number of positive electric charges and a large number of negative electric charges. These positive and negative charges nearly exactly cancel each other out. It is customary to perform calculations using a fictitious charge, which is the net difference between the positive and negative charge, averaged over the region of space. This fictitious charge usually cannot be identified with any specific positive or negative particle.

A magnetic current is another example of a fictitious source that is often used. It is generally assumed that magnetic monopoles and magnetic currents do not exist (while electric monopoles and electric currents do exist). Nevertheless, it is known how to mathematically relate electric currents to equivalent magnetic currents to produce the same electromagnetic waves. The use of magnetic sources is widely accepted, and has proven very useful for certain types of calculations. Sometimes, it is convenient to use a source that is a particular combination of electric and magnetic sources. A distribution of sources over some region of space can also be used as a source. The terms "sources" and "physical sources" are used herein to include all types of actual and/or fictitious sources.

A physical source at one location typically produces a disturbance that propagates to a sensor (or tester) at another location. Mathematically, the interaction between a source and a tester is often expressed as a coupling coefficient (usually as a complex number having a real part and an imaginary part). The coupling coefficients between a number of sources and a number of testers is usually expressed as an array (or matrix) of complex numbers. Embodiments of this invention include efficient methods for the computation of these complex numbers, for the storing of these complex numbers, and for computations using these complex numbers.

The so-called Method of Moments (MoM) is an example of a numerical analysis procedure that uses interactions between source functions and testing functions to numerically solve a problem that involves finding an unknown function (that is, where the solution requires the determination of a function of one or more variables). The MoM is used herein by way of example and not as a limitation. One skilled in the art will recognize that the MoM is one of many types of numerical techniques used to solve problems, such as differential equations and integral equations, where one of the unknowns is a function. The MoM is an example of a class of solution techniques wherein a more difficult or unsolvable problem is broken up into one or more interrelated but simpler problems. Another example of this class of solution techniques is Nystrom's method. The simpler problems are solved, in view of the known interrelations between the simpler problems, and the solutions are combined to produce an approximate solution to the original, more difficult, problem.

For example, Figure 1A shows a wire or rod 100 having a physical property (e.g., a current, a temperature, a stress, a voltage, a vibration, a displacement, etc.) along its length. An expression for the physical property is shown as an unknown function $I(\ell)$. The problem is to calculate $I(\ell)$ using the MoM or a similar "divide and conquer" type of technique. By way of example, in many physical problems involving temperature, vibration, or electrical properties, etc. $I(\ell)$ will be described by an integral equation of the form:

$$E(\overline{R}) = \int I(\ell)G(\ell,\overline{R})d\ell$$

Where $G(\ell,\overline{R})$ is known everywhere and $E(\overline{R})$ is known for certain values of $\overline{R}$. In many circumstances, $G(\ell,\overline{R})$ is a Green's function, based on the underlying physics of the problem, and the value of $E(\overline{R})$ is known only at boundaries (because of known boundary conditions). The above equation is usually not easily solved because $I(\ell)$ is not known, and thus the integration cannot be performed. The above integral equation can be turned into a differential equation (by taking the derivative of both sides), but that will not directly provide a solution. Regardless of whether the above equation is expressed as an integral equation or a differential equation, the equation can be numerically solved for

$I(\ell)$ by creating a set of simpler but interrelated problems as described below (provided that $G(\ell,\overline{R})$ possesses certain mathematical properties known to those of skill in the art).

As shown in Figure 1B, in order to compute a numerical approximation for $I(\ell)$, the wire 100 is first divided up into four segments 101-104, and basis function $f_1(\ell)$, $f_2(\ell)$, and $f_3(\ell)$ are selected. In Figure 1B the basis functions are shown as triangular-shaped functions that extend over pairs of segments. The unknown function $I(\ell)$ can then be approximated as:

$$I(\ell) \approx I_1 f_1(\ell) + I_2 f_2(\ell) + I_3 f_3(\ell)$$

where I_1, I_2, and I_3 are unknown complex constants. Approximating $I(\ell)$ in this manner transforms the original problem from one of finding an unknown function, to a problem of finding three unknown constants. The above approximation for $I(\ell)$ is inserted into the original integral equation above to yield:

$$E(\overline{R}) = \int I_1 f_1(\ell) G(\ell,\overline{R}) d\ell + \int I_2 f_2(\ell) G(\ell,\overline{R}) d\ell + \int I_3 f_3(\ell) G(\ell,\overline{R}) d\ell$$

The above integrals can now be performed because the functional form of the integrands are all known ($G(\ell,\overline{R})$ is determined by the problem being solved, the functions $f_i(\ell)$ were selected, and the constants I_1, I_2 and I_3 can be moved outside the integrals). However, this does not yet solve the problem because the values of I_1, I_2 and I_3 are still unknown.

Fortunately, as indicated above, the value of $E(\overline{R})$ is usually known at various specific locations (e.g., at boundaries). Thus, three equations can be written by selecting three locations $\overline{R}_1$, $\overline{R}_2$, $\overline{R}_3$, where the value of $E(\overline{R})$ is known. Using these three selected locations, the above equation can be written three times as follows:

$$E(\overline{R}_1) = I_1 \int f_1(\ell)G(\ell,\overline{R}_1)d\ell + I_2 \int f_2(\ell)G(\ell,\overline{R}_1)d\ell + I_3 \int f_3(\ell)G(\ell,\overline{R}_1)d\ell$$
$$E(\overline{R}_2) = I_1 \int f_1(\ell)G(\ell,\overline{R}_2)d\ell + I_2 \int f_2(\ell)G(\ell,\overline{R}_2)d\ell + I_3 \int f_3(\ell)G(\ell,\overline{R}_2)d\ell$$
$$E(\overline{R}_3) = I_1 \int f_1(\ell)G(\ell,\overline{R}_3)d\ell + I_2 \int f_2(\ell)G(\ell,\overline{R}_3)d\ell + I_3 \int f_3(\ell)G(\ell,\overline{R}_3)d\ell$$

Rather than selecting three specific locations for $E(\overline{R})$, it is known that the accuracy of the solution is often improved by integrating known values of $E(\overline{R})$ using a weighting function over the region of integration. For example, assuming that $E(\overline{R})$ is known along the surface of the wire 100, then choosing three weighting functions $g_1(\ell)$, $g_2(\ell)$, and $g_2(\ell)$, the desired three equations in three unknowns can be written as follows (by multiplying both sides of the equation by $g_i(\ell)$ and integrating):

$$\int E(\ell')g_1(\ell')d\ell' = I_1 \int\int f_1(\ell)g_1(\ell')G(\ell,\ell')d\ell d\ell' + I_2 \int\int f_2(\ell)g_1(\ell')G(\ell,\ell')d\ell d\ell' + I_3 \int\int f_3(\ell)g_1(\ell')G(\ell,\ell')d\ell d\ell'$$
$$\int E(\ell')g_2(\ell')d\ell' = I_1 \int\int f_1(\ell)g_2(\ell')G(\ell,\ell')d\ell d\ell' + I_2 \int\int f_2(\ell)g_2(\ell')G(\ell,\ell')d\ell d\ell' + I_3 \int\int f_3(\ell)g_2(\ell')G(\ell,\ell')d\ell d\ell'$$
$$\int E(\ell')g_3(\ell')d\ell' = I_1 \int\int f_1(\ell)g_3(\ell')G(\ell,\ell')d\ell d\ell' + I_2 \int\int f_2(\ell)g_3(\ell')G(\ell,\ell')d\ell d\ell' + I_3 \int\int f_3(\ell)g_3(\ell')G(\ell,\ell')d\ell d\ell'$$

Note that the above double-integral equations reduce to the single-integral forms if the weighting functions $g_i(\ell)$ are replaced with delta functions. As an alternative, the calculation can be done using such delta functions, such as when Nystrom's method is used.

The three equations in three unknowns can be expressed in matrix form as:

V=ZI

or

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} Z_{11} & Z_{12} & Z_{13} \\ Z_{21} & Z_{22} & Z_{23} \\ Z_{31} & Z_{32} & Z_{33} \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix}$$

where

$$V_i = \int E(\ell')g_i(\ell')d\ell'$$

and

$$Z_{ij} = \int\int f_j(\ell)g_i(\ell')G(\ell,\ell')d\ell d\ell$$

Solving the matrix equation yields the values of I_1, I_2, and I_3. The values I_1, I_2, and I_3 can then be inserted into the equation $I(\ell) \approx I_1 f(\ell) + I_2 f_2(\ell) + I_3 f_3(\ell)$ to give an approximation for $I(\ell)$. If the basis functions are triangular functions as shown in Figure 1B, then the resulting approximation for $I(\ell)$ is a piecewise linear approximation as shown in Figure 1C. The I_i are the unknowns and the V_i are the conditions (typically, the V_i are knowns). Often there are the same number of conditions as unknowns. In other cases, there are more conditions than unknowns or less conditions than unknown.

The accuracy of the solution is largely determined by the shape of the basis functions, by the shape of the weighting functions, and by the number of unknowns (the number of unknowns usually corresponds to the number of basis functions).

Unlike the Moment Method described above, some techniques do not use explicit basis functions, but, rather, use implicit basis functions or basis-like functions. For example, Nystrom's method produces a numerical value for an integral using values of the integrand at discrete points and a quadrature rule. Although Nystrom's method does not explicitly use an expansion in terms of explicit basis functions, nevertheless, in a physical sense, basis functions are still being used (even if the use is implicit). That is, the excitation of one unknown produces some reaction throughout space. Even if the computational method does not explicitly use a basis function, there is some physical excitation that produces approximately the same reactions. All of these techniques are similar, and one skilled in the art will recognize that such techniques can be used with the present invention. Accordingly, the term "basis function" will be used herein to include such implicitly used basis functions. Similarly, the testers can be implicitly used.

When solving most physical problems (e.g., current, voltage, temperature, vibration, force, etc), the basis functions tend to be mathematical descriptions of the source of some physical disturbance. Thus, the term "source" is often used to refer to a basis function. Similarly, in physical problems, the weighting functions are often associated with a receiver or sensor of the disturbance, and, thus, the term "tester" is often used to refer to the weighting functions.

As described above in connection with Figures 1A-1C, in numerical solutions, it is often convenient to partition a physical structure or a volume of space into a number of smaller pieces and associate the pieces with one or more sources and testers. In one embodiment, it is also convenient to partition the structure of (or volume) into regions, where each region contains a group of the smaller pieces. Within a given region, some number of sources is chosen to describe with sufficient detail local interactions between sources and testers within that region. A similar or somewhat smaller number of sources in a given region is generally sufficient to describe interactions between sources in the source region and testers in the regions relatively close by. When the appropriate sources are used, an even smaller number of sources is often sufficient to describe interactions between the source region and testers in regions that are not relatively close by (i.e., regions that are relatively far from the source region).

Embodiments of the present invention include methods and techniques for finding composite sources. Composite sources are used in place of the original sources in a region such that a reduced number of composite sources is needed to calculate the interactions with a desired accuracy.

In one embodiment, the composite sources for a first region are the same regardless of whether the composite sources in the first region are interacting with a second region, a third region, or other regions. The use of the same composite sources throughout leads to efficient methods for factoring and solving the interaction matrix.

Considering the sources in the first region, one type of source is the so-called multipole, as used in a multipole expansion. Sources like wavelets are also useful. In some cases wavelets allow a reduced number of composite sources to be used to describe interactions with distant regions. However, there are disadvantages to wavelet and multipole approaches. Wavelets are often difficult to use, and their use often requires extensive modifications to existing or proposed computer programs. Wavelets are difficult to implement on non-smooth and non-planar bodies.

Multipole expansions have stability problems for slender regions. Also, while a multipole expansion can be used for describing interactions with remote regions, there are severe problems with using multipoles for describing interactions within a region or between spatially close regions. This makes a factorization of the interaction matrix difficult. It can be very difficult to determine how to translate information in an interaction matrix into a wavelet or multipole representation.

Figure 2 is a flowchart that illustrates a compression technique 200 for compressing an interaction matrix by combining groups of sources and groups of testers into composite sources and testers. The use of composite sources and composite testers allows the original interaction matrix to be transformed into a block sparse matrix having certain desirable properties.

Embodiments of the present invention include a technique for computing and using composite sources to provide compression of an interaction matrix by transforming the interaction matrix into a block sparse matrix. The present technique is compatible with existing and proposed computer programs. It works well even for rough surfaces and irregular grids of locations. For a given region, the composite sources allow computation of a disturbance (e.g., radiation) produced by the source throughout a desired volume of space. A reduced number of these composite sources is sufficient to calculate (with a desired accuracy) disturbances at other relatively distant regions. This method of compressing interaction data can be used with a variety of computational methods, such as, for example, an LU (Lower Triangular Upper triangular) factorization of a matrix or as a preconditioned conjugate gradient iteration. In many cases, the computations can be done while using the compressed storage format.

Figure 2 is a flowchart 200 illustrating the steps of solving a numerical problem using composite sources. The flowchart 200 begins in a step 201 where a number of original sources and original testers are collected into groups, each group corresponding to a region. Each element of the interaction matrix describes an interaction (a coupling) between a source and a tester. The source and tester are usually defined, in part, by their locations in space. The sources and testers are grouped according to their locations in space. In one embodiment, a number of regions of space are defined. A reference point is chosen for each region. Typically the reference point will lie near the center of the region. The sources and testers are grouped into the regions by comparing the location of the source or tester to the reference point for each region. Each source or tester is considered to be in the region associated with the reference point closest to the location. (For convenience, the term "location" is used hereinafter to refer to the location of a source or a tester.)

Other methods for grouping the sources and testers (that is, associating locations with regions) can also be used. The process of defining the regions is problem-dependent, and in some cases the problem itself will suggest a suitable set of regions. For example, if the sources and testers are located on the surface of a sphere, then curvilinear-square regions are suggested. If the sources and testers are located in a volume of space, then cubic regions are often useful. If the sources and testers are located on a complex three-dimensional surface, then triangular patch-type regions are often useful.

Generally the way in which the regions are defined is not critical, and the process used to define the regions will be based largely on convenience. However, it is usually preferable to define the regions such that the locations of any region are relatively close to each other, and such that there are relatively few locations from other regions close to a given region. In other words, efficiency of the compression algorithm is generally improved if the regions are as isolated from one another as reasonably possible. Of course, adjacent regions are often unavoidable, and when regions are adjacent to one another, locations near the edge of one region will also be close to some locations in an adjacent region. Nevertheless, the compression will generally be improved if, to the extent reasonably possible, regions are defined such that they are not slender, intertwining, or adjacent to one another. For example, Figure 3 illustrates a volume of space partitioned into a rectangular box 300 having eleven regions A through K corresponding to reference points 301-311. In come cases, the regions will not overlap. In one embodiment, the regions overlap in places. A source (or a tester) located within an overlap of two (or more) regions can be associated with both of those two (or more) regions. As a result, such sources (and testers) can be used in building composite sources associated with two (or more) regions.

As shown in Figure 2, after the step 201 the process advances to a step 202. In the step 202, the unknowns are renumbered, either explicitly or implicitly, so that locations within the same region are numbered consecutively. It is simpler to continue this description as if the renumbering has actually been done explicitly. However, the following analysis can also be performed without explicit renumbering. A computer program can also be written either with the renumbering, or without renumbering. With the appropriate bookkeeping, the same result may be achieved either way.

The term "spherical angles" is used herein to denote these angles. One skilled in the art will recognize that if a two-dimensional problem is being solved, then the spherical angles reduces to a planar angle. Similarly, one skilled in the art will recognize that if a higher-dimensional problem is being solved (such as, for example, a four dimensional space having three dimensions for position and one dimension for time) then the term spherical angle denotes the generalization of the three-dimensional angle into four-dimensional space. Thus, in general, the term spherical angle is used herein to denote the notion of a "space-filling" angle for the physical problem being solved.

After renumbering, the process advances to a block 203 where one or more composite sources for each region are determined. If there are p independent sources within a region, then q composite sources can be constructed (where $q \leq p$). The construction of composite sources begins by determining a relatively dense set of far-field patterns (usually described in a spherical coordinate system) at relatively large distances from the region. As used herein, far-field refers to the field in a region where the field can be approximated in terms of an asymptotic behavior. For example, in one embodiment, the far-field of an antenna or other electromagnetic radiator includes the field at some distance from the antenna, where the distance is relatively larger than the electrical size of the antenna.

A far-field pattern using a dense collection is constructed for each independent source. In the present context, dense means to avoid having any overly-large gaps in the spherical angles used to calculate the set of disturbances. Dense also means that if the disturbance is represented by a vector, then each vector component is represented. For example, for a scalar problem, one can choose p spherical angles. These angles are typically substantially equally spaced, and the ranges of angles include the interaction angles occurring in the original interaction matrix (if all of the interactions described in the original matrix lie within a plane, then one can choose directions only within that plane rather than over a complete sphere).

The far-field data is stored in a matrix s having p columns (one column for each source location within the region), and rows associated with angles. This matrix often has as many rows as columns, or more rows than columns. While each source is logically associated with a location in a given region, these sources are not necessarily located entirely within that region. While each source corresponds to a location (and each location is assigned to a region), sources that have a physical extent can extend over more than one region. The entries in the matrix s can be, for example, the field quantity or quantities that emanate from each source. It is desirable that the field quantity is chosen such that when it (or they) are zero at some angle then, to a desired approximation, all radiated quantities are zero at that angle. While it is typically desirable that the angles be relatively equally spaced, large deviations from equal spacing can be acceptable. One method for producing far-field data is to use the limiting form of the data for relatively large distances. Another method is to pick a point within the region, and to use the data for some relatively large distance or distances from that point, in the direction of each angle. Relatively large can be defined as large relative to the size of that region. Other methods can also be used.

These composite sources are in the nature of equivalent sources. A smaller number of composite sources, compared to the number of sources they replace, can produce similar disturbances for regions of space removed from the region occupied by these sources.

As described above, sources are collected into groups of sources, each group being associated with a region. For each group of sources, a group of composite sources is calculated. The composite source is in the nature of an equivalent source that, in regions of space removed from the region occupied by the group in replaces, produces a far-field (disturbance) similar to the field produced by the group it replaces. Thus, a composite source (or combination of composite sources) efficiently produces the same approximate effects as the group of original sources at desired spherical angles and at a relatively large distance. To achieve a relatively large distance, is it often useful to use a limiting form as the disturbance goes relatively far from its source.

Each composite source is typically a linear combination of one or more of the original sources. A matrix method is used to find composite sources that broadcast strongly and to find composite sources that broadcast weakly. These composite sources are constructed from the original sources. The matrix method used to find composite sources can be a rank-revealing factorization such as singular value decomposition. For a singular value decomposition, the unitary transformation associated with the sources gives the composite sources as a linear combination of sources.

Variations of the above are possible. For example, one can apply the singular value decomposition to the transpose of the *s* matrix. One can employ a Lanczos Bi-diagonalization, or related matrix methods, rather than a singular value decomposition. There are other known methods for computing a low rank approximation to a matrix. Some examples of the use of Lanczos Bidiagonalization are given in Francis Canning and Kevin Rogovin, "Fast Direct Solution of Standard Moment-Method Matrices," IEEE AP Magazine, Vol. 40, No. 3, June 1998, pp. 15-26.

There are many known methods for computing a reduced rank approximation to a matrix. A reduced rank approximation to a matrix is also a matrix. A reduced rank matrix with *m* columns can be multiplied by any vector of length *m*. Composite sources that broadcast weakly are generally associated with the space of vectors for which that product is relatively small (e.g., in one embodiment, the product is zero or close to zero). Composite sources that broadcast strongly are generally associated with the space of vectors for which that product is not necessarily small.

Composite sources can extend over more than one region. In one embodiment, this is achieved by using the technique used with Malvar wavelets (also called local cosines) to extend Fourier transforms on disjoint intervals to overlapping orthogonal functions. This results in composite sources associated with one region overlapping the composite sources associated with another (nearby) region. In one embodiment, this feature of sources associated with one region overlapping sources associated with a nearby region can be achieved by choosing regions that overlap and creating composite sources using these overlapping regions.

Persons of ordinary skill in the art know how near-field results are related to far-field results. A relationship between near-field and far-field can be used in a straightforward way to transform the method described above using far-field data into a method using near-field data. Note that, the "far-field" as used herein is not required to correspond to the traditional $2d^2/\lambda$ far-field approximation. Distances closer than $2d^2/\lambda$ can be used (although closer distances will typically need more composite sources to achieve a desired accuracy). A distance corresponding to the distance to other physical regions is usually far enough, and even shorter distances can be acceptable.

Once composite sources are found, the process advances to a step 204 where composite testers are found. Composite testers are found in a manner analogous to the way that composite sources are found. Recall that composite sources are found using the way in which sources of the interaction matrix "broadcast" to distant locations. Composite testers are found using the way in which the testers of the interaction matrix "receive" from a dense group of directions for a distant disturbance. It is helpful if the received quantity or quantities which are used include relatively all field quantities, except (optionally) those which are very weakly received. For example, when receiving electromagnetic radiation from a distant source, the longitudinal component is approximately zero and can often be neglected. A matrix R describing how these testers receive is formed. A matrix method is used to construct composite testers that receive strongly and testers that receive weakly. The matrix method can be a rank-revealing factorization such as singular value decomposition. A singular value decomposition gives the composite testers as a linear combination of the testers which had been used in the original matrix description.

An alternative method for determining how testers receive can be used in creating the matrix R. The direction of motion of the physical quantity in the tester (if any) can be reversed. This corresponds to the concept of time reversal. When certain common conventions are used, this can be accomplished by replacing the tester by its complex conjugate. Then, the tester is used as if it were a source, and its effect is determined as was done for sources. Then, this effect undergoes a time reversal. In some cases, that time reversal can be accomplished by taking a complex conjugate. While these time reversal steps are often desirable, often they are not essential, and good results can be achieved by omitting them.

Once composite sources and testers have been found, the process advances to a step 205 or to a step 215 where the interaction matrix is transformed to use composite sources and testers. The steps 205 and 215 are alternatives. Figure 4 shows an example of an interaction matrix 400 having 28 unknowns (28 sources and 28 testers) grouped into five physical regions (labeled I-V). The shaded block 401 of the matrix 400 represents the interaction for sources in the fourth region (region IV) and testers in the second region (region II). The interaction of a pair of regions describes a block in the interaction matrix 400. The blocks of the transformed matrix can be computed at any time after the composite functions for their source and tester regions are both found. That is, the block 401 can be computed after composite sources for region IV and testers for region II are found.

The step 215 of Figure 2 shows one method for computing all of the blocks in the matrix 400 by computing the entries for these blocks using the original sources and testers. Then, the process advances to an optional step 216 where these blocks are transformed into a description in terms of the composite sources and composite testers.

One advantage of using composite sources and testers is that many entries in the transformed matrix will be zero. Therefore, rather than transforming into a description using composite modes, the step 205 shows calculation of the transformed block directly using the composite sources and composite testers (without first calculating the block using the original sources and testers). In other words, the composite sources are used as basis functions, and the composite testers are used as weighting functions. Within each block, entries that are known *au priori* to be zero (or very small) are not calculated. Those skilled in the art will understand that there are still more equivalent methods for creating the transformed matrix. As an example, a portion of the transformed matrix can be computed, and then that portion and known properties about such matrices can be used to find the remainder of the matrix.

Further savings in the storage required are possible. After each block has been transformed, only the largest elements are kept. No storage needs to be used for the elements that are approximately zero. Many types of block structures, including irregular blocks and multilevel structures, can also be improved by the use of this method for storing a block sparse matrix. This will usually result in a less regular block structure. As an alternative, it is also possible to store a portion of the interaction data using composite sources and testers and to store one or more other portions of the data using another method.

The non-zero elements of the interaction matrix typically occur in patterns. After either the step 205 or the step 216, the process advances to a step 206 where the interaction matrix is reordered to form regular patterns. For a more uniform case, where all of the regions have the same number of sources, the resulting transformed matrix T is shown in Figure 5. Figure 5 shows non-zero elements as shaded and zero elements as unshaded. If only a compressed storage scheme is desired, the process can stop here. However, if it is desired to calculate the inverse of this matrix, or something like its LU (lower-upper triangular) factorization, then a reordering can be useful.

The rows and columns of the interaction matrix can be reordered, to produce a matrix T^ in the form shown in Figure 6. This permutation moves the composite sources that broadcast strongly to the bottom of the matrix, and it moves the composite testers which receive strongly to the right side of the matrix. The interaction between composite sources and composite testers is such that the sizes of the matrix elements can be estimated *au priori*. A matrix element that corresponds to an interaction between a composite source that radiates strongly and a composite tester that receives strongly will be relatively large. A matrix element that corresponds to an interaction between a composite source that radiates strongly and a composite tester that receives weakly will be relatively small. Similarly, a matrix element that corresponds to an interaction between a composite source that radiates weakly and a composite tester that receives strongly will be relatively small. A matrix element that corresponds to an interaction between a composite source that radiates weakly and a composite tester that receives weakly will be very small.

The permuted matrix T^ often will tend to be of a banded form. That is, the non-zero elements down most of the matrix will tend to be in a band near the diagonal. For a matrix of this form, there are many existing sparse-matrix LU factorers and other matrix solvers, that work well. The order shown in Figure 6 is one example. The LU decomposition of the matrix T^ can be computed very rapidly by standard sparse matrix solvers. The matrices L and U of the LU decomposition will themselves be sparse. For problems involving certain types of excitations, only a part of the matrices L and U will be needed, and this can result in further savings in the storage required.

After reordering, the process 200 advances to a step 207 where the linear matrix problem is solved. The matrix problem to be solved is written as:

T^ G=H

where the vector H represents the excitation and the vector G is the desired solution for composite sources. The excitation is the physical cause of the sound, temperature, electromagnetic waves, or whatever phenomenon is being computed. If the excitation is very distant (for example, as for a plane wave source), H will have a special form. If the vector H is placed vertically (as a column vector) alongside the matrix of Figure 6, the bottom few elements of H alongside block 602, will be relatively large, and the remaining elements of H will be approximately equal to zero. The remaining elements of H are approximately zero because the composite testers separate the degrees of freedom according to how strongly they interact with a distant source.

When T^ is factored by LU decomposition, then:

T^ = LU;

L U G = H;

and this is solved by the following two-step process;

Step I: Find X in L X = H

Step II: Find G in U G = X

The matrix L is a lower triangular matrix (meaning elements above its diagonal are zero). It follows immediately from this that if only the bottom few elements of H are non-zero, then only the bottom elements of X are non-zero. As a consequence, only the bottom right portion of L is needed to compute G. The remaining parts of L were used in computing this bottom right portion, but need not be kept throughout the entire process of computing the LU decomposition. This not only results in reduced storage, but also results in a faster computation for Step I above.

If only the far-field scattered by an object needs to be found, then further efficiencies are possible. In that case, it is only necessary to find the bottom elements of G, corresponding to the bottom non-zero elements of H. This can be done using only the bottom right portion of the upper triangular matrix U. This results in efficiencies similar to those obtained for L.

For other types of excitations, similar savings are also possible. For example, for many types of antennas, whether acoustic or electromagnetic, the excitation is localized within one active region, and the rest of the antenna acts as a passive scatterer. In that case, the active region can be arranged to be represented in the matrix of Figure 6 as far down and as far to the right as possible. This provides efficiencies similar to those for the distant excitation.

A permutation of rows and a permutation of columns of the matrix T of Figure 5 brings it to the matrix T^ of Figure 6. These permutations are equivalent to an additional reordering of the unknowns. Thus, a solution or LU decomposition of the matrix T^ of Figure 6 does not immediately provide a solution to the problem for the original data. Several additional steps are used. These steps involve: including the effects of two permutations of the unknowns; and also including the effect of the transformation from the original sources and testers to using the composite sources and composite testers.

Direct methods (such as LU decomposition) and iterative methods can both be used to solve the matrix equation herein. An iterative solution, with the compressed form of the matrix, can also be used with fewer computer operations than in the prior art. Many iterative methods require the calculation of the product of a matrix and a vector for each iteration. Since the compressed matrix has many zero elements (or elements which can be approximated by zero), this can be done more quickly using the compressed matrix. Thus, each iteration can be performed more quickly, and with less storage, than if the uncompressed matrix were used.

The compressed format of T^ has an additional advantage. In many cases, there is a way to substantially reduce the number of iterations required, resulting in further increases in speed. For example, in the method of conjugate gradients, the number of iterations required to achieve a given accuracy depends on the condition number of the matrix. (The condition number of a matrix is defined as its largest singular value divided by its smallest.) Physical problems have a length scale, and one interpretation of these composite sources and composite testers involves length scales. These composite sources and composite testers can be described in terms of a length scale based on a Fourier transform. This physical fact can be used to improve the condition number of the matrix and therefore also improve the speed of convergence of the iterative method.

A composite source is a function of spatial position, and its Fourier transform is a function of "spatial frequency." Composite sources that broadcast weakly tend to have a Fourier transform that is large when the absolute value of this spatial frequency is large. There is a correlation between how large this spatial frequency is and the smallness of the small singular values of the matrix. This correlation is used in the present invention to provide a method to achieve convergence in fewer iterations.

Two matrices, P^R and P^L are defined as right and left preconditioning matrices to the compressed matrix. Each column of the compressed matrix is associated with a composite source. This composite source can be found using a matrix algebra method, such as a rank-revealing factorization (e.g., singular value decomposition and the like). The rank-revealing factorization method provides some indication of the strength of the interaction between that composite source and other disturbances. For example, using a singular value decomposition, the associated singular value is proportional to this strength. The diagonal matrix P^R is constructed by choosing each diagonal element according to the interaction strength for the corresponding composite source. The diagonal element can be chosen to be the inverse of the square root of that strength. Similarly, the diagonal matrix P^L can be constructed by choosing each diagonal element according to the interaction strength for its associated composite tester. For example, the diagonal element can be chosen to be the inverse of the square root of that strength.

If the compressed matrix is called T, then the preconditioned matrix is

$P = P^L T P^R$

The matrix P will often have a better (i.e., smaller) condition number than the matrix T. There are many iterative methods that will converge more rapidly when applied to the preconditioned matrix P rather than to T.

One embodiment of the composite source compression technique is used in connection with the computer program NEC2. This program was written at Lawrence Livermore National Laboratory during the 1970s and early 1980s. The NEC2 computer program itself and manuals describing its theory and use are freely available over the Internet. The following development assumes NEC2 is being used to calculate the electromagnetic fields on a body constructed as a wire grid.

NEC2 uses electric currents flowing on a grid of wires to model electromagnetic scattering and antenna problems. In its standard use, NEC2 generates an interaction matrix, herein called the Z matrix. The actual sources used are somewhat complicated. There is at least one source associated with each wire segment. However, there is overlap so that one source represents current flowing on more than one wire segment. NEC2 uses an array CURX to store values of the excitation of each source.

Figure 10 is a flowchart 1000 showing the process of using NEC2 with composite sources and composite testers. The flowchart 1000 begins at a step 1001 where the NEC2 user begins, as usual, by setting up information on the grid of wires and wire segments. The process then advances to a step 1002 to obtain from NEC2 the number of wire segments, their locations (x,y,z coordinates), and a unit vector $\hat{t}$ for each segment. The vector $\hat{t}$ is tangent along the wire segment, in the direction of the electric current flow on the wire segment.

Next, in a step 1003, the wire grid is partitioned into numbered regions. A number of reference points are chosen. The reference points are roughly equally spaced over the volume occupied by the wire grid. Each wire segment is closest to one of these reference points, and the segment is considered to be in the region defined by the closest reference point. In one embodiment, the number of such points (and associated regions) is chosen as the integer closest to the square root of N (where N is the total number of segments). This is often an effective choice, although the optimum number of points (and associated regions) depends on many factors, and thus other values can also be used. For a set of N segments, each wire segment has an index, running from 1 to N.

After the step 1003, the process advances to a step 1004 where the wires are sorted by region number. After sorting, the numbering of the wires is different from the numbering used by NEC2. Mapping between the two numbering systems is stored in a permutation table that translates between these different indexes for the wire segments. Using this new numbering of segments, an array "a" is created, such that $a(p)$ is the index of the last wire segment of the p^{th} region (define $a(0) = 0$ for convenience).

After renumbering, the process advances to a step 1005 where composite sources and composite testers for all regions are calculated. Source region p corresponds to unknowns $a(p\text{-}1) + 1$ through $a(p)$ in the ordering. Define M as $M = a(p) - a(p - 1)$. Choose M directions substantially equally spaced throughout three-dimensional space. In other words, place M roughly equally spaced points on a sphere, and then consider the M directions from the center of the sphere to each point. The order of the directions is unimportant. One convenient method for choosing these points is similar to choosing points on the earth. For example, choose the North and South poles as points. A number of latitudes are used for the rest of the points. For each chosen latitude, choose points equally spaced at a number of longitudes. This is done so that the distance along the earth between points along a latitude is approximately the same as the distance between the latitude lines holding the points. It is desirable that the points are equally spaced. However, even fairly large deviations from equal spacing are tolerable.

Now generate a matrix A of complex numbers with $2M$ rows and M columns. For $m = 1$ to M and for $n = 1$ to M., compute elements of this matrix two at a time: the element at row m and column n and also the element at row $m + M$ and column n. To compute these two elements, first fill the NEC2 array CURX with zero in every position. Then, set position $a(p\text{-}1) + n$ of CURX to unity. A value of unity indicates that only source number $a(p\text{-}1) + n$ is excited. This source is associated with the wire segment of that number, even though it extends onto neighboring segments. The matrix Z is defined in terms of these same sources. Then, call the NEC2 subroutine `CABC(CURX)`. The subroutine `CABC` generates a different representation of the source, but the same representation that the NEC2 subroutine `FFLD` uses. This representation is automatically stored within NEC2. The m^{th} direction previously chosen can be described in spherical coordinates by the pair of numbers `(Theta, Phi)`. After calculating `Theta` and `Phi`, the NEC2 subroutine `FFLD(Theta,Phi,ETH,EPH)` is called. `Theta` and `Phi` are inputs, as are the results from `CABC`. The outputs from `FFLD` are the complex numbers `ETH` and `EPH`. `ETH` and `EPH` are proportional to the strengths of the electric field in the far-field (far away from the source) in the theta and phi directions respectively. `ETH` is placed in row m and column n, (m,n), of A. `EPH` is placed at row $m+M$ and column n of A. Alternatively, there are other ways to compute the numbers `ETH` and `EPH` produced by `FFLD`. For example, it will apparent to one of ordinary skill in the art that the subroutine `FFLD` can be modified to produce an answer more quickly by making use of the special form of the current, since most of the entries in the current are zero.

Next, a singular value decomposition of A is performed, such that:

$$A = U\,D\,V^h$$

where U and V are unitary matrices, and D is a diagonal matrix. The matrix U will not be used, so one can save on computer operations by not actually calculating U. The matrix V has M rows and M columns. Since these calculations are performed for the p^{th} region, the square matrix d_p^R is defined by

$$d_p^R = V$$

The reason for this choice comes from the fact that

$$A\,V = U\,D$$

and that each successive columns of the product UD tends to become smaller in magnitude. They become smaller because U is unitary and the singular values on the diagonal of D decrease going down the diagonal.

Next, assemble an N by N block diagonal matrix D^R. That is, along the diagonal the first block corresponds to d_p^R with $p = 1$. Starting at the bottom right corner of that block, attach the block for $p = 2$, etc., as shown in Figure 7.

Next a similar procedure is followed to find the block diagonal matrix D^L. For each region p, a matrix A is filled as before. However, this time this region is considered as receiving rather than as transmitting. Again A will have $2M$ rows and M columns, where $M = a(p) - a(p-1)$. Again there are M directions, but now those are considered to be the receiving directions.

To understand what is to be put into A, it is instructive to note how the NEC2 computer program defines the interaction matrix Z. When used with wire grid models, the sources radiate electric and magnetic fields. However, it is the electric field reaching another segment that is used in NEC2. Each matrix element of Z is computed by computing the component of that electric field which is in the direction of the tangent to the wire segment.

For the pair of numbers (m,n), where $m = 1,\ldots, M$ and $n = 1,\ldots,M$, the matrix entries for A at (m,n) and $(m+M,n)$ are calculated as follows. Compute a unit vector $\hat{k}$ in the m^{th} direction. Find the unit vector tangent to segment number n, and call it $\hat{t}$. The position of the center of wire segment number n is found and is designated as the vector X. Then compute the vector

$$\bar{f} = (\hat{t} - (\hat{k} \cdot \hat{t})\hat{k})e^{j2\pi\hat{k}\cdot\bar{X}/\lambda}$$

where the wavelength is given by λ (NEC2 uses units where $\lambda = 1$).

Note that the Green's function for this problem has a minus sign in the exponential, and the foregoing expression does not. This is because the direction of $\hat{k}$ is outward, which is opposite to the direction of propagation of the radiation.

For problems in electromagnetics, the physical wavelength λ is greater than zero. If a problem in electrostatics is being solved instead, electrostatics can be considered as the limit when the wavelength becomes arbitrarily large. The complex exponential above can then be replaced by unity. Also, for electrostatics, the relevant field quantity can be longitudinal (meaning f would be parallel to $\hat{k}$).

For this value of m (and associated direction $\hat{k}$), spherical coordinates define two directions called the theta and the phi directions. These directions are both perpendicular to the direction of $\hat{k}$. Compute the components of f in each of these directions, and designate them as `fTheta` and `fPhi`. These are complex numbers. Then place `fTheta` in row m and column n of A and place `fPhi` in row $m + M$ and column n of A.

The matrix A is a matrix of complex numbers. Take the complex conjugate of A, (A*), and perform a singular value decomposition on it, such that:

$$A^* = U\,D\,V^h$$

Now define the left diagonal block for region p, d_p^L, as

$$d_p^L = V^h$$

The superscript h on V, indicates Hermitian conjugate. The definition of the blocks for the right side did not have this Hermitian conjugate. From these diagonal blocks, assemble an N by N matrix D^L similar to the way D^R is assembled. The motivation for these choices is partly that the matrix DU^h has rows that tend to become smaller. Further, it is expected that the Green's function used in creating Z has properties similar to the far-field form used in creating A^t. The formula

$$V^h A^t = D\,U^h$$

shows that $V^h A^t$ will also have successive rows that tend to become smaller. The choices described above suggest that successive rows of each block of the compressed matrix will also have that property.

It should be noted that the matrix A, whether used for the right side or for the left side, can be filled in other ways as well. For example, with an appropriate (consecutive in space) ordering of the angles, A can be made as an M by M matrix by using theta polarization (`fTheta`) values for one angle and phi polarization values (`fPhi`) for the next. Usually, it is desirable that A does not leave large gaps in angle for any component of the electric field, which is important far from the source or receiver.

In performing the singular value decompositions for the right and left sides, singular values are found each time. Figures 8 and 9 show the singular values found for blocks of size 67 by 93 and 483 by 487, respectively. These calculations were done for a wire grid model with NEC2. The singular values are plotted in terms of how many orders of magnitude they are smaller than the largest singular value, and this is called "Digits of Accuracy" on the plots. Figures 8 and 9 show the accuracy that is achieved when truncating to a smaller number of composite sources or composite testers for regions that are relatively far apart. For regions that are closer together, the desired accuracy often requires the information from more composite sources and composite testers to be kept.

After computing composite sources and composite testers, the process advances to a step 1006 where a new matrix T, which uses the composite sources and testers associated with D^L and D^R, is computed. The matrix is T given by the equation

$$T = D^L \; Z \; D^R$$

T can be efficiently generated by using the numbering of the wire segments developed herein (rather than the numbering used in NEC2). The matrix Z is computed by NEC2 and renumbered to use the numbering described herein. Note that a block structure has been overlaid on Z and T. This block structure follows from the choice of regions. Figure 4 shows one example of a block structure. Block $\{p,q\}$ of the matrix T, to be called $T\{p,q\}$, is the part of T for the rows in region number p and the columns in region number q. The formula for T given above is such that $T\{p,q\}$ only depends on $Z\{p,q\}$. Thus, only one block of Z at a time needs to be stored.

In the step 1006, T is assembled one block at a time. For each block of T, first obtain from NEC2 the corresponding block of Z. The wire segments within a block are numbered consecutively herein (NEC2 numbers them differently). Thus, first renumber Z using the renumber mapping from step 1004, and then perform the calculation:

$$T\{p,q\} = d_p^{\,L}\, Z\{p,q\} d_q^{\,R}$$

Many of the numbers in $T\{p,q\}$ will be relatively small. An appropriate rule based on a desired accuracy is used to choose which ones can be approximated by zero. The remaining non-zero numbers are stored. Storage associated with the zero-valued elements of $T\{p,q\}$ and of $Z\{p,q\}$ can be released before the next block is calculated. The top left portion of $T\{p,q\}$ has matrix elements which will be kept. Anticipating this, the calculation speed can be increased by not calculating either the right portion or the bottom portion of $T\{p,q\}$.

The matrix T is a sparse matrix, and it can be stored using an appropriate data structure for a sparse matrix. For a matrix with N_z non-zero elements, an array $Z_z(i)$ for $i = 1,\ldots,N_z$ can be used, where $Z_z(i)$ is the complex value of the i^{th} matrix element. There are two integer valued arrays, $I_z(i)$ and $J_z(i)$ for $i = 1,...,N_z$. $I_z(i)$ gives the row number where the i^{th} matrix element occurs in T and $J_z(i)$ its column number.

After calculation of T, the process proceeds to a process block 1007 where the rows and columns of the matrix T are reordered to produce a matrix T^. The matrix T is reordered into a matrix T^ so that the top left corner of every block of T^ ends up in the bottom right corner of the whole matrix. The T^ form is more amenable to LU factorization. Figure 5 shows an example of a matrix T, and Figure 6 shows an example of a matrix T^ after reordering. One embodiment uses a solver that has its own reordering algorithms thus negating the need for an explicit reordering from T to T^.

After reordering, the process advances to a step 1008 where the matrix T^ is passed to a sparse matrix solver, such as, for example, the computer program "Sparse," from the Electrical Engineering Department of University of California at Berkeley. The program Sparse can be used to factor the matrix T^ into a sparse LU decomposition.

NEC2 solves the equation

$$J = Z^{-1} E$$

for various vectors E. In Figure 10, the solution of the above matrix equation is done in steps 1009-1016 or, alternatively, in steps 1017-1023. The sequence of steps 1009-1016 is used with a matrix equation solver that does not provide reordering. The sequence of steps 1017-1023 is used with a matrix equation solver that does provide reordering.

In the step 1009, the vector E is computed by NEC2. Then, in the step 1010, the elements of E are permutated (using the same permutation as that used in the step 1004) to produce a vector E'. This permutation is called the region permutation. Next, in the step 1011, E' is expressed in terms of composite testers by multiplying E' by D^L, giving D^LE'. Then, in the step 1012, the same permutation used in the step 1007 is applied to D^LE' to yield (D^LE')^. (This permutation is called the solver permutation.) Then, in the step 1013, a matrix equation solver (such as, for example, the solver known as "SPARSE") is used to solve for the vector Y^ from the equation

$$T\text{^}(Y\text{^})=(D^L E')\text{^}$$

Then, in the step 1014, the inverse of the solver permutation is applied to Y^ to yield Y. In the subsequent step 1015, J' is computed from the equation

$J'=D^{R}Y$

In the subsequent, and final, step 1016, the inverse of the region permutation is applied to J' to yield the desired answer J.

Alternatively, the embodiment shown in steps 1017-1023 is conveniently used when the matrix equation solver provides its own reordering algorithms, thus eliminating the need to reorder from T to T^ (as is done in the step 1007 above). In the step 1017, a reordering matrix solver is used to solve the matrix T. In the subsequent step 1018, the vector E is computed by NEC2. Then, in the step 1019, the elements of E are permutated using the region permutation to produce a vector E'. Then, in the step 1020, $D^{L}E'$ is computed. The process then proceeds to the step 1021 where the equation

$TY=D^{L}E'$

is solved for Y. After Y is computed, the process advances to the step 1022 where J' is calculated from the equation

$J' = D^{R}Y$

Finally, in the step 1023, the inverse of the region permutation is applied to J' to yield the desired answer J.

Many matrix elements are made small by this method. Figures 11 and 12 show before and after results for a problem using a wire grid model in NEC2, with a matrix Z of size 2022 by 2022 and a block of size 67 by 93. Figure 11 shows the magnitudes of the matrix elements before changing the sources and testers, meaning it shows a 67 by 93 block of the renumbered Z. Figure 12 shows this same block of T. The matrix T has a regular structure wherein the large elements are in the top left corner. This is a general property of the transformed matrix. For larger blocks, the relative number of small matrix elements is even better.

The algorithms expressed by the flowchart shown in Figure 2 can be implemented in software and loaded into a computer memory attached to a computer processor to calculate, for example, propagation of energy, pressure, vibration, electric fields, magnetic fields, strong nuclear forces, weak nuclear forces, etc. Similarly, the algorithms expressed by the flowchart shown in Figure 10 can be implemented in software and loaded into a computer memory attached to a computer processor to calculate, for example, electromagnetic radiation by an antenna, electromagnetic scattering, antenna properties, etc.

One embodiment includes a method for manipulating, factoring and inverting interaction data and related data structures efficiently and with reduced storage requirements. One embodiment also includes methods that are easily tuned for a specific computer's architecture, and that allow that computer to process instructions at a high rate of speed. For example, when data and instructions are already available in a computer's high speed cache when an instruction occurs that needs this information, then that instruction may proceed without a relatively long wait for that data to be moved. This allows instructions to be executed at a higher rate of speed.

Methods have been described above for compressing interaction data. This data often is stored as an array, which can be used in equations. Such interaction data often has many elements which are approximately zero and which can be ignored. The pattern of the location of zeros can be called the sparseness pattern. A class of sparseness patterns occurs for interaction data before and/or after the compression methods described above, and also for other data For example, it applies to data where there is a relatively large amount of data to describe each entity, and relatively less data being passed between these entities. These entities might be, for example, computers connected by a network or business organizations within a larger company or within a consortium. The invention relates to efficient methods for using such data.

An array of data can, for example, be used to multiply or be divided into data. For example, sometimes it is desired to find the inverse of a matrix or to divide either a vector or a matrix by a matrix. One embodiment includes efficient methods for quickly finding the inverse and/or dividing. While many methods are known for performing such operations, this invention relates to finding highly efficient methods for a particular sparseness structure. Such methods should ideally require relatively few operations, use operations which execute quickly on computers, and should require the storage of relatively few numbers.

The matrix structure shown in Figure 5 is a particular sparse structure. This figure is not meant as a limitation; rather it is meant as a schematic guide. The actual structure can differ significantly from this and the method described here can nevertheless be useful. However, this idealized structure can be used as an aid in developing a method which is more general than for just this structure.

Often there is a need to find a solution for Y in the matrix problem

$$T\,Y = V \qquad (1)$$

where T is a matrix and Y and V are vectors. These vectors and the matrix can contain elements that can be multiplied and divided, including but not limited to elements such as real numbers and complex numbers. While methods for solving the above equation are have been presented above, an alternative embodiment is as follows. This alternative embodiment provides an alternative method for performing step 1017 in Figure 10. The step 1017 is described in the flow chart as "Solve T using a reordering matrix solver." In one embodiment, the present alternative method avoids the "reordering" step, and can be used to replace reordering matrix solvers such as the package "Sparse" from the University of California at Berkeley.

There are several desirable attributes of a method for solving this equation. First, the number of computational operations needed in the solution should be reduced.. Second, the computational operations should be arranged so as to run efficiently (e.g., quickly) the desired computing platforms. That is, it should be possible for the desired computing platforms to execute many operations per second. Third, the matrix T is sparse, meaning many elements of T are zero, and the number of non-zero elements of T is generally smaller than the total number of elements of T. It is generally desirable that the solution for Y should be found using as few numbers as possible so that the number of matrix elements that must be stored and accessed is small.

One known direct method for finding Y is to compute the LU decomposition of T. When this is done, elements that are zero in T can give rise to non-zero elements in the corresponding position in L or U. Here, L represents a lower triangular matrix and U represents an upper triangular matrix. Embodiments have been given above where the rows and columns of T are permuted in order to reduce this "fill in" of non-zero elements. However, the present embodiment introduces a different approach which often provides all three of the desirable properties listed above. This approach involves applying the LU decomposition method to sub-matrices within T rather than to the elements of T. These sub-matrices generally contain elements of T.

Figure 5 shows a block structure within T. That is, the columns of T can be naturally grouped into ranges of columns. The rows of T can also be grouped into ranges of rows. As an example, the matrix T might be created in a way that naturally associates a group of columns and/or of rows with some physical region. This occurred for some matrices described above. A block or sub-matrix within T is the portion of T corresponding to one range of columns and one range of rows of T. T is composed of the collection of these non-overlapping blocks. Since each such block is a sub-matrix, the rules for matrix multiplication, division, addition and subtraction are well known. These rules are described in elementary mathematics books.

This method can be applied to less regular block structures. For example, Figure 5 shows a structure where each block has the same width as other blocks and the same height as other blocks. It also shows blocks where there height is the same as their width. This is not meant as a limitation, but is used solely as an illustration of one example case. Also, some matrices T may not have a structure like that shown in Figure 5, but a permutation of their rows and columns can produce such a structure. The method herein can be applied to a permuted matrix, and computations using that permuted matrix will give the desired answer.

The standard formulas for LU decomposition of a matrix of numbers can also be applied to a matrix of sub-matrices. Two sub-matrices can be multiplied just as two numbers can be multiplied, provided the dimensions of the sub-matrices are properly related. However, this condition is satisfied when the standard formula for LU decomposition is applied to sub matrices. The multiplication of matrices is not commutative, so care must be taken in writing the order of the factors for the LU decomposition in terms sub-matrices. However, with this care the standard formula for numbers applies to sub-matrices also.

For the structure shown in Figure 5, consider the standard LU factorization method applied to the numbers of T without applying any permutation. The fill in of non-zero elements would be significant. Choose any row of T, and find the left most non-zero element of that row. From that element moving to the right until reaching the diagonal, every element generally will be non-zero in the factor L. Similarly, starting with a non-zero element above the diagonal there generally is fill in below it until reaching the diagonal.

For the idealized structure shown in Figure 5, this fill in can be avoided by a factorization which is applied to the blocks of T rather than to the elements of T. This result is due to a block structure of the matrix, such as the example matrix shown in Figure 5. Some notation will be useful in describing this. When Equation (1) is described by its block structure the result is:

$$\begin{bmatrix} T_{1,1} & T_{1,2} & \dots & T_{1,m} \\ T_{2,1} & T_{2,2} & \dots & T_{2,m} \\ \dots & \dots & \dots & \dots \\ T_{m,1} & T_{m,2} & \dots & T_{m,m} \end{bmatrix} \bullet \begin{bmatrix} Y_1 \\ Y_2 \\ \dots \\ Y_m \end{bmatrix} = \begin{bmatrix} V_1 \\ V_2 \\ \dots \\ V_m \end{bmatrix} \qquad (2)$$

Here, $T_{2,m}$ does not represent one number within the matrix T. Rather, this particular block represents a sub-matrix within T, for region m interacting with region 2. The structure of Equation (2) is analogous to the structure that results when Equation (1) is written in terms of the numbers within the matrix T. However, here the elements in the matrix in Equation (2) are themselves matrices of numbers. These matrices are blocks from the matrix T. A block LU factorization using this block structure is a factorization of T into a block lower triangular matrix L and a block upper triangular matrix U. In one embodiment, the diagonal blocks of L are identity matrices. The LU factorization can be written

$$L\,U = T \qquad (3)$$

This has a block structure, which for this embodiment is:

$$\begin{bmatrix} I & 0 & \dots & 0 \\ A_{2,1} & I & \dots & 0 \\ \dots & \dots & \dots & \dots \\ A_{m,1} & A_{m,2} & \dots & I \end{bmatrix} \bullet \begin{bmatrix} B_{1,1} & B_{1,2} & \dots & B_{1,m} \\ 0 & B_{2,2} & \dots & B_{2,m} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & B_{m,m} \end{bmatrix} = \begin{bmatrix} T_{1,1} & T_{1,2} & \dots & T_{1,m} \\ T_{2,1} & T_{2,2} & \dots & T_{2,m} \\ \dots & \dots & \dots & \dots \\ T_{m,1} & T_{m,2} & \dots & T_{m,m} \end{bmatrix} \quad (4)$$

Here, each I is an identity matrix. The sub-matrices in any column of sub-matrices (i.e. sub-matrices with the same second index) all have the same number of columns of elements as each other. However, sub-matrices from different block columns can have differing numbers of columns of elements. Similarly, for sub-matrices from the same row of sub-matrices, they each have the same number of rows within them. The elements of L (given by $A_{i,j}$) and the elements of U (given by $B_{i,j}$) can be found from the algorithm:

```
For j = 1 to m
{
    { for i=1 to j   $\left[ B_{i,j} = T_{i,j} - \sum_{k=1}^{i-1} A_{i,k} B_{k,j} \right]$ }        (5)

    {for i=j+1 to m  $\left[ \begin{matrix} \tilde{A}_{i,j} = T_{i,j} - \sum_{k=1}^{j-1} A_{i,k} B_{k,j} \\ A_{i,j} = \tilde{A}_{i,j} \bullet B_{j,j}^{-1} \end{matrix} \right]$ }
}
```

Notice that the multiplication by the inverse of $B_{j,j}$ is done on the right side. The multiplication of sub-matrices is not commutative. Reversing the order of operations of products in Equation (5) will generally give incorrect results.

It is usually desirable to perform computations so that sparse storage is used and so that the number of internal computations is minimized and so that these computations execute quickly on computers. Figure 13 shows an idealized view of the sparse storage within blocks of A and B. In particular, a block of B, $B_{i,j}$, is generally sparse when i is not equal to j. This figure shows that a block $\tilde{A}_{i,j}$ is also sparse. This is a result that follows from the particular structure shown in Figure 13 and related structures. This result is not in general true for all sparse matrix structures.

A first particular embodiment of an improved method can now be described. The operations in Equation (5) are reordered so that all computed blocks for one block row below the diagonal are found before beginning operating on the next block row. While operating on a block row, $B_{i,j}$ for $i<j$ and for $i=j$, and also $A_{i,j}$ and $\tilde{A}_{i,j}$ for $i>j$ are stored. When moving on to succeeding rows, $A_{i,j}$ will not be retained, but the other quantities are retained. Thus, the quantities which are retained are sparse. This modification to the algorithm of Equation (5) gives an embodiment which is:

```
For i = 1 to m
 {
     For j = 1 to i-1                                          (6)
 {
      $\left[\begin{array}{l} \tilde{A}_{i,j} = T_{i,j} - \sum_{k=1}^{j-1} A_{i,k} B_{k,j} \\ A_{i,j} = \tilde{A}_{i,j} \bullet B_{j,j}^{-1} \end{array}\right]$
      delete $T_{i,j}$
 }
For j = i to m
 {
      $\left[ B_{i,j} = T_{i,j} - \sum_{k=1}^{i-1} A_{i,k} B_{k,j} \right]$
      delete $T_{i,j}$
 }
compute and store $B_{i,j}^{-1}$
For j = 1 to i-1
      {
      delete $A_{i,j}$
      }
 }
```

This embodiment illustrates a general property of the LU decomposition. Many different orders of operations are possible, provided that each quantity $A_{i,j}$ or $B_{i,j}$ is computed before it is used. Other variations will be evident to those experienced in this field. For example, it is possible to use an LDM decomposition rather than an LU decomposition. Typically, D then is a block diagonal matrix and L and M have identity matrices on their diagonal blocks. Further variations are also evident, for example one might compute (LD) D^{-1} (DM) and store (LD) rather than L and store (DM) rather than M.

The embodiment of Equation (6) proceeds by finding the quantities $A_{i,j}$ and $B_{i,j}$ within row "i" of L and U. Then, "i" is increase by one and this is repeated until "i" equals m. Similarly, an alternative embodiment might find these quantities in a different order within L and U. However, for such an embodiment the quantities A and $\tilde{A}$ would be handled differently.

The general method described here involves replacing the individual operations on matrix elements by block operations involving relatively small sub-matrices. The non-zero elements within a block can be considered as part of a small rectangular sub-block which is just large enough to contain these non-zero elements. In one embodiment this can be treated as a full sub-block. This sub-block is generally smaller than the block, so even treating this sub-block as full and storing it as such can leave the block as a whole still sparse. This allows a method which applies to more general sparse structures than that shown in Figure 5. In terms of Figure 5, the small square regions of non-zero numbers within larger blocks are shown as square regions. When a less-regular region of a block contains non-zero numbers, it is possible to find a larger regular region which contains the non-zero numbers, and to apply this algorithm to that more regular region. Often, that regular region will be rectangular.

In this embodiment, computations can be performed using full rectangular sub-blocks (within larger blocks) and performing computations with very efficient optimized packages, such as level 2 and level 3 "BLAS" (basic linear algebra subroutine) packages. This generally allows a computer to execute computations at a high speed. Often, this can result in a speed improvement of nearly a factor of ten, or more.

In addition, a very reduced operation count can be achieved by this general method. Figure 13 shows that the product $A_{i,k}$ $B_{k,j}$ results in a sparse block. The operation count to compute this product is especially small since only the leftmost columns of $A_{i,k}$ are used in this computation. For the number of non-zero elements illustrated in Figure 13 this product requires 64 times fewer operations than would be required for a computation with full blocks.

Note that Figure 13 shows square blocks for purposes of illustration only. In general, these blocks need not be square. Nevertheless, the basic algorithm is not affected. For example, when computing the matrix product $A_{i,k}$ $B_{k,j}$ the number of rows and columns of $A_{i,k}$ may not be equal and the number of rows and columns of $B_{k,j}$ may not be equal. However, the number of columns of $A_{i,k}$ will equal the number of rows of $B_{k,j}$ so there is no difficulty in performing the matrix product.

The specific embodiments described above have the advantage that the "back substitution" and "forward substitution" steps are actually faster when using $\tilde{A}_{i,j}$ rather than A. Define D to be a diagonal matrix with block j down the diagonal $B_{j,j}^{-1}$, then

$$LU = \tilde{L}DU =$$

$$\begin{bmatrix} B_{1,1} & 0 & \dots & 0 \\ \tilde{A}_{2,1} & B_{2,2} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \tilde{A}_{m,1} & \tilde{A}_{m,2} & \dots & B_{m,m} \end{bmatrix} \bullet \begin{bmatrix} B_{1,1}^{-1} & 0 & \dots & 0 \\ 0 & B_{2,2}^{-1} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & B_{m,m}^{-1} \end{bmatrix} \bullet \begin{bmatrix} B_{1,1} & B_{1,2} & \dots & B_{1,m} \\ 0 & B_{2,2} & \dots & B_{2,m} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & B_{m,m} \end{bmatrix} \quad (7)$$

The three factors here can be used to compute solutions to the associated linear equation by using the same methods that were used to compute these factors. This results in a sparse algorithm that executes operations quickly on many computers and that has a reduced operation count.

The decomposition of Equation (7) provides an algorithm for solving the linear equation, Equation (1), for each vector V. The basic algorithm uses forward substitution for $\tilde{L}$ and back substitution for U, just as these methods are used with the standard LU decomposition. Naturally, a block form of these algorithms is used here.

In a further description of the embodiment illustrated in Equation (6), note that the computation involving the inverse of each block $B_{j,j}$ can be performed by computing either the LU decomposition of this block and using that or by actually computing the inverse (possibly from its LU decomposition) of the block and using that. For this embodiment, one can choose to actually use the inverse. This can have advantages (such as a reduced operation count) when multiplying this inverse times a sparse matrix block. Note, that this inverse can be computed in a stable way using (e.g., by using an LU decomposition of the block), computed with pivoting, as an intermediate step. This adds stability to the overall computation. Pivoting within each block this way will often be sufficient for stability, without pivoting among blocks.

After performing the factorization of Equation(6) the first step in solving Equation(1) for each vector E is to solve for the vector X in

$$\tilde{L} X = E \mathrm{V} \qquad (8)$$

The vector V is composed of m sub vectors. According to the standard forward substitution algorithm (applied to sub vectors), X can be found from the algorithm:

```
For p = 1 to m
  {
        $X_p = B_{p,p}^{-1}[V_p - \sum_{i=1}^{p-1} \tilde{A}_{p,i} X_i]$        (9)
  }
```

The next step is to solve for F in the equation

$$D F = X \tag{10}$$

Where D is the block diagonal matrix used in Equation (7). Again, F is composed of m sub vectors. The vector F can be found from the algorithm

```
For p = 1 to m
  {
        $F_p = B_{p,p} X_p$        (11)
  }
```

Finally, Y is also composed of m sub vectors, which can be found from the standard back substitution algorithm applied to sub vectors, which is the algorithm

```
For p = m to 1, step –1  (i.e. decrease p by one each time)
  {
        $Y_p = B_{p,p}^{-1}[F_p - \sum_{i=p+1}^{m} B_{p,i} Y_i]$        (12)
  }
```

In Equations (9) and (12), as is standard practice when the sum is empty (When p=1 and when p=m respectively), that sum is replaced by a zero vector.

The portion of the algorithm given by Equations (8-11) can be simplified further. Equation (13) below gives an equivalent computation that is simpler because it does not require a multiplication by $B_{p,p}$. This algorithm will execute quicker, and it has the further advantage that it does not require that $B_{p,p}$ be stored for all p up to p equals m.

```
For p = 1 to m
  {
        $F_p = [V_p - \sum_{i=1}^{p-1} \tilde{A}_{p,i} X_i]$                    (13)
        $X_p = B_{p,p}^{-1} F_p$
  }
```

The algorithm described by Equations (11-12) or by Equations (13) and Equations (12) allows one to find Y from V and the factorization of the matrix T. These algorithms can be implemented using level 2 BLAS, since they involve matrix–vector operations. They also can be applied to a number of vectors playing the role of V to compute a number of solutions Y at one time. Since a number of vectors V, placed one after the other, is a matrix, each sub vector V would then be replaced by a sub matrix. This would allow the computation to be done using level 3 BLAS, which performs matrix – matrix operations. This allows a computer to perform operations at an even faster rate (more computations per second).

Figure 14 shows results for the speed actually achieved by the embodiment of Equation (6). These results are for a personal computer (PC) with a one GigaHertz (GHz) central processor. Matrices were created for interaction data which was compressed according to the method described above. The plusses on the figure show the time achieved for these six matrices. The solid line shows the time generally taken by standard methods on a full (uncompressed, and where all elements are non-zero) matrix of the same size. When these methods are optimized by using efficient machine-specific BLAS routines, the times generally improve to that shown by the dotted line. The plusses all indicate a significantly better time than that shown here for other methods.

The algorithms in the above disclosure can be implemented in software and loaded into a computer memory attached to a computer processor to calculate, for example, propagation of energy, pressure, vibration, electric fields, magnetic fields, strong nuclear forces, weak nuclear forces, etc. Similarly, the algorithms can be implemented in software and loaded into a computer memory attached to a computer processor to calculate, for example, electromagnetic radiation by an antenna, electromagnetic scattering, antenna properties, etc.

Although the foregoing has been a description and illustration of specific embodiments of the invention, various modifications and changes can be made thereto by persons skilled in the art without departing from the scope and spirit of the invention. For example, in addition to electromagnetic fields, the techniques described above can also be used to compress interaction data for physical disturbances involving a heat flux, an electric field, a magnetic field, a vector potential, a pressure field, a sound wave, a particle flux, a weak nuclear force, a strong nuclear force, a gravity force, etc. The techniques described above can also be used for lattice gauge calculations, economic forecasting, state space reconstruction, and image processing (e.g., image formation for synthetic aperture radar, medical, or sonar images). Accordingly, the invention is limited only by the claims that follow.

Appendix A

A.1 Listing of file BlockwiseFactor.h

```
#ifndef __BLOCKWISE_FACTOR_H__
#define __BLOCKWISE_FACTOR_H__

//----------------------------------------
//----------------------------------------
//  Copyright (c) 2002 Francis X. Canning
//  all rights reserved
//----------------------------------------
//----------------------------------------

#include "KDevilBase.h"
#include "FortranMatrix.h"
#include "FullLU.h"

class BlockwiseFactor:virtual public KDevilBase
{
private:
  //data needed for factoring....
  FortranMatrix ***theFortranBlocks;

  //some stat stuff....
  f_complex **blockwise_diagnolElementsLocal;
  f_complex *blockwise_diagnolElementsGlobal;
  char *blockwise_diagnol_filename;

  //storage space needed while factoring...
  FortranMatrix *mylittleworker;
  FortranMatrix **myBigWorker;
  FullLU_NRC *tempNRCLUmatrix;
  f_complex **tempLUmatrix, *mylittlevector;
  int *tempswapdata;
  int *workerwidth, *workerheight;

  //storage space needed for solving....
  f_complex **yVec, **rhsVec;

public:
  virtual ~BlockwiseFactor() {}
  BlockwiseFactor(char *filename, char *diagname=NULL);

  // so we can specify what order to create the blocks in
  virtual void CalculateBlocks(void);

  //will call the sparse factoring routine
  // SparseFcator will also re-order the SparseMatrix to making
factoring easier
```

```
  // this re-ordering is _INTERNAL_ to the sparsematrix stuff, but uses
the data in
  // SingularValues to determine how to re-order..
  virtual void SparseFactor(void);

  // wrapper to SparseSolver
  // will do the re-oreding as necessary on the vector,
  // here the re-ording is the sutff done in SparseFactor()
  virtual void SparseSolve(f_complex *);

  // called to initially create SparseMatrix structure
  virtual void InitSparseMatrix(void);

  // does whatever cleanup (if any at all) necessary for the
sparesematrix
  virtual void DeInitSparseMatrix(void);

  //initilizes the SparseSolver
  // will be called after the creation of VLeft/VRight but before the
  // calcualtion of any blocks
  virtual void InitSparseSolver(void);

  //De-initilizes the SparseSolver
  virtual void DeInitSparseSolver(void);

  virtual int SparseAddBlock(int blockrow, int blockcol, int numrows,
int numcols, f_complex **data);


  virtual void GenerateSparseReordering(void);

};


#endif
```

A.2 Listing of file BlockwiseFactor.c

```
#include "BlockwiseFactor.h"
//----------------------------------------
//----------------------------------------
//  Copyright (c) 2002 Francis X. Canning
//  all rights reserved
//----------------------------------------
//----------------------------------------

BlockwiseFactor::BlockwiseFactor(char *initFile, char *diagFile)
{
  blockwise_diagnol_filename=diagFile;

}
```

```
void BlockwiseFactor::CalculateBlocks(void)
{
  int blockrow, blockcol;

  //force the creation of blocks so that we do whole rows at a time...
  for(blockrow=1;blockrow<=nblocks;blockrow++)
  {
    cout << "\n" << blockrow <<")"; cout.flush();
    for(blockcol=1;blockcol<=nblocks;blockcol++)
    {
      CreateBlock(blockrow,blockcol);
      cout <<"."; cout.flush();
    }
  }
}


void BlockwiseFactor::SparseFactor(void)
{
  k_real biggest, smallest, temp;
  int bigindex, smallindex;

  //nothing to do!
  //we did the sparse factor, while creating the matrix!

  //however we can print the diagnol elements out, along with their
ratios and such...
  if (blockwise_diagnol_filename!=NULL)
  {
    ofstream outfile(blockwise_diagnol_filename);


    bigindex=smallindex=1;
    biggest=smallest=sqabs(blockwise_diagnolElementsGlobal[1]);
    for(int row=1;row<=N;row++)
      {
      outfile << blockwise_diagnolElementsGlobal[row];
      temp=sqabs(blockwise_diagnolElementsGlobal[row]);

      if (temp>=biggest)
        {
          bigindex=row;
          biggest=temp;
        }
      if (temp<=smallest)
        {
          smallest=temp;
          smallindex=row;
        }
      }


    cout << "\nLargestDiagnol Element at " << bigindex << " is "
       << blockwise_diagnolElementsGlobal[bigindex];
    cout << "\nSmallestDiagnol Element at " << smallindex << " is "
```

```
        <<  blockwise_diagnolElementsGlobal[smallindex];

      cout << "\nTheir ratio is " << sqrt(biggest/smallest) << "\n";

    }


  }

  void BlockwiseFactor::GenerateSparseReordering(void)
  {
    //do nothing!
  }

  void BlockwiseFactor::SparseSolve(f_complex *rhs)
  {
    int blockrow, localrow, k;
    f_complex negativeone, positiveone, zero;
    int incr_one, blockheight, blockwidth;

    negativeone=-1.0;
    positiveone=1.0;
    zero=0.0;
    incr_one=1;

    //first make a nice blockwise rhs vector...
    //we are NOT copying data, just setting pointers!
    rhsVec[1]=rhs;
    for(blockrow=2;blockrow<=nblocks;blockrow++)
      rhsVec[blockrow]=rhsVec[blockrow-1]+BlockSizes[blockrow-1];


    //we now do the modified LU-solving stuff...
    //we first do:
    //     rhsVec[i] -= sum_{k=1}^{i-1} theFortranBlocks[i][k]*yVec[k] )
    //we then do:
    //     yVec[i] = theFortranBlocks[i][i]*rhsVec[i]
    for(blockrow=1;blockrow<=nblocks;blockrow++)
    {
      //first modify rhsVec[blockrow],,,
      for(k=1;k<blockrow;k++)
      {
        if (theFortranBlocks[blockrow][k]!=NULL)
        theFortranBlocks[blockrow][k]->MatrixVectorBlas(-
1,1,yVec[k],rhsVec[blockrow]);
      }
      //now      apply      theFortranBlocks[blockrow][blockrow]      to
rhsVec[blockrow].....
      //putting result into yVec...
      ASSERT(theFortranBlocks[blockrow][blockrow]!=NULL);
      theFortranBlocks[blockrow][blockrow]-
>ApplyToVector(rhsVec[blockrow],yVec[blockrow]);

    }
  //now do the back-solver...
  //we are to do:
  //   rhsVec[i]   -=   sum_{k=i+1}^{nblocks}   theFortranBlocks[i][k]   *
yVec[k]
  // then we do:
  // yVec[i]  = FortranBlocks[i][i] * rhsVec[i]
  for(blockrow=nblocks;blockrow>=1;blockrow--)
  {
    //first modify rhsVec[blockrow]....
    for(k=blockrow+1;k<=nblocks;k++)
    {
      if (theFortranBlocks[blockrow][k]!=NULL)
      theFortranBlocks[blockrow][k]->MatrixVectorBlas(-1,1,yVec[k],
rhsVec[blockrow]);
    }

    //     now     apply     theFortranBlocks[blockrow][blockrow]     to
rhsVec[blockrow].....
    // putting result into yVec
    ASSERT(theFortranBlocks[blockrow][blockrow]!=NULL);
    theFortranBlocks[blockrow][blockrow]-
>ApplyToVector(rhsVec[blockrow],yVec[blockrow]);
  }

  //now  copy  yVec  contents  to  rhsVec...  as  rhsVec  is  the  returned
data...
  for(blockrow=1;blockrow<=nblocks;blockrow++)
    for(localrow=1;localrow<=BlockSizes[blockrow];localrow++)
      rhsVec[blockrow][localrow]=yVec[blockrow][localrow];

  //we are done, as rhsVec <--> rhs



}

void BlockwiseFactor::DeInitSparseMatrix(void)
{
  int i,j;

  for(i=1;i<=nblocks;i++)
  {
    for(j=1;j<=nblocks;j++)
      if (theFortranBlocks[i][j]!=NULL)
      delete theFortranBlocks[i][j];

    theFortranBlocks[i]++;
    delete theFortranBlocks[i];
  }

  theFortranBlocks++;
  delete theFortranBlocks;
  delete mylittleworker;
  for(j=1;j<nblocks;j++)
    delete myBigWorker[j];

  myBigWorker++;
  delete myBigWorker;

  workerwidth++; delete workerwidth;
  workerheight++; delete workerheight;
  tempswapdata++; delete tempswapdata;
  mylittlevector++; delete mylittlevector;
  delete tempNRCLUmatrix;
  for(j=1;j<=largestblocksize;j++)
    {
      tempLUmatrix[j]++;
      delete tempLUmatrix[j];
    }
  tempLUmatrix++;
  delete tempLUmatrix;
  blockwise_diagnolElementsLocal++;
  delete blockwise_diagnolElementsLocal;
  blockwise_diagnolElementsGlobal++;
  delete blockwise_diagnolElementsGlobal;

}

void BlockwiseFactor::InitSparseMatrix(void)
{
  int i,j;

  //allocate the pointer arrays for the FortranBlocks,
  theFortranBlocks=new FortranMatrix**[nblocks];
  theFortranBlocks--;

  for(i=1;i<=nblocks;i++)
  {
    theFortranBlocks[i]=new FortranMatrix*[nblocks];
    theFortranBlocks[i]--;
    for(j=1;j<=nblocks;j++)
      theFortranBlocks[i][j]=NULL;
  }

  //create temporary storage...
  mylittleworker=new FortranMatrix(largestblocksize, largestblocksize);
  myBigWorker=new FortranMatrix*[nblocks-1];
  myBigWorker--;
  for(j=1;j<nblocks;j++)
  {
    myBigWorker[j]=new FortranMatrix(largestblocksize, BlockSizes[j]);
  }
  workerwidth=new int[nblocks-1];
  workerwidth--;
  workerheight=new int[nblocks-1];
  workerheight--;
  tempLUmatrix=new f_complex*[largestblocksize];
  tempLUmatrix--;
  for(j=1;j<=largestblocksize;j++)
  {
    tempLUmatrix[j]=new f_complex[largestblocksize];
    tempLUmatrix[j]--;
  }
  tempswapdata=new int[largestblocksize];
  tempswapdata--;

  // when we change the data in tempLUmatrix we modify
  // the values in tempNRCLUmatrix and vica versa
  tempNRCLUmatrix=new     FullLU_NRC(largestblocksize,     tempLUmatrix,
tempswapdata);
  mylittlevector=new f_complex[largestblocksize];
  mylittlevector--;


  blockwise_diagnolElementsGlobal=new f_complex[N]-1;
  blockwise_diagnolElementsLocal=new f_complex*[nblocks]-1;

  for(j=1;j<=nblocks;j++)
    blockwise_diagnolElementsLocal[j]=blockwise_diagnolElementsGlobal  -
1 + BlockStarts[j];
}


void BlockwiseFactor::InitSparseSolver(void)
{
  int blocknum;

  rhsVec=new f_complex*[nblocks];
  rhsVec--;

  yVec=new f_complex*[nblocks];
  yVec[0]=new f_complex[N];

  yVec[0]--;
  yVec--;

  for(blocknum=2;blocknum<=nblocks;blocknum++)
    yVec[blocknum]=BlockSizes[blocknum-1]+yVec[blocknum-1];
}

void BlockwiseFactor::DeInitSparseSolver(void)
{
}


int BlockwiseFactor::SparseAddBlock(int blockrow, int blockcol,
                              int numrows, int numcols, f_complex **data)
{
  int localrow,localcol, maxindex;
  int maxrow_needed, maxcol_needed;
  int k, blockheight, blockwidth;
  int kwidth, kblockheight, kblockwidth;


  blockheight=BlockSizes[blockrow];
  blockwidth=BlockSizes[blockcol];



  //initialize mylittleworker as all 0's
  for(localrow=1;localrow<=BlockSizes[blockrow];localrow++)
    for(localcol=1;localcol<=BlockSizes[blockcol];localcol++)
      mylittleworker->Element(localrow,localcol)=0;

  //dump the _ALL_ the data into mylittleworker
  // we'll truncate elements later...
  for(localrow=1;localrow<=numrows;localrow++)
    for(localcol=1;localcol<=numcols;localcol++)
      mylittleworker-
>Element(localrow,localcol)=data[localrow][localcol];


  if(blockrow>blockcol) // below diagnol
    maxindex=blockcol;
  else //at diagnol or above diagnol
    maxindex=blockrow;

  //we now do:
  //mylittleworker   -=     sum_{k=1}^{maxindex-1}   (   myBigWorker[k]   *
Block[k][blockcol] )

  for(k=1;k<maxindex;k++)
  {
    //  if  theFortranBlocks[k][blockcol]  is  NULL,  then  is  is  a  zero
block
    //  if  workerwidth[k]  is  0  then  myBigWorker[k]  represents  an  empty
block
    if (theFortranBlocks[k][blockcol]!=NULL && workerwidth[k]!=0)
      {

      //we only multiply as necessary
      if(workerwidth[k]< theFortranBlocks[k][blockcol]->getheight())
        kwidth=workerwidth[k];
      else
        kwidth=theFortranBlocks[k][blockcol]->getheight();

      //we only add what the trucnated blocks will create....
      ASSERT( blockwidth >=theFortranBlocks[k][blockcol]->getwidth());
      kblockwidth=theFortranBlocks[k][blockcol]->getwidth();

      ASSERT(blockheight>=workerheight[k]);
      kblockheight=workerheight[k];

      ASSERT(k<blockcol);
      ASSERT(kblockheight<=blockheight);
      ASSERT(kblockwidth<=blockwidth);
      // makes the assigment:
      //           mylittleworker          <----           -1          *
myBigWorker[k]*theFortranBlocks[k][blockol]
      //                              + 1.0 * mylittleworker
      theFortranBlocks[k][blockcol]-
>TruncatedOnRightMultMatrix(kblockheight,
                                              kblockwidth,kwidth,
                                              -1.0, myBigWorker[k],
                                              1.0, mylittleworker);


      }
    else
      {
      //do nothing! as one of the matrices is empty!
      }
  }



  //now decide what is worth keeping....
  if (blockrow==blockcol) //if on diagnol, keep it all!
  {
    maxrow_needed=maxcol_needed=blockwidth;
  }
  else
  {

    maxrow_needed=maxcol_needed=0;
    for(localrow=1;localrow<=blockheight;localrow++)
      for(localcol=1;localcol<=blockwidth;localcol++)
      {
        k_real temp_magsq;

        temp_magsq= sqabs(mylittleworker->Element(localrow,localcol));
        if (temp_magsq>= actualtolsq)
          {
            if (localrow>maxrow_needed)
            maxrow_needed=localrow;
            if (localcol>maxcol_needed)
            maxcol_needed=localcol;
          }
      }
  } // of else.

  if (maxcol_needed!=0)
    {
      //now store into the FortranBlocks.....
      theFortranBlocks[blockrow][blockcol]=new
FortranMatrix(maxrow_needed,maxcol_needed);
      for(localrow=1;localrow<=maxrow_needed;localrow++)
      for(localcol=1;localcol<=maxcol_needed;localcol++)
                                  theFortranBlocks[blockrow][blockcol]-
>Element(localrow,localcol)=mylittleworker->Element(localrow,localcol);
```

```
    }
  else // empty block!
    {
      theFortranBlocks[blockrow][blockcol]=NULL;
      ASSERT(maxrow_needed==0);
      ASSERT(maxcol_needed==0);
    }

  if(blockrow==blockcol)        //we        need        to        invert
theFortranBlocks[blockrow][blockcol]
  {
    ASSERT(theFortranBlocks[blockrow][blockcol]!=NULL);
    //we'd like to grab the diagnol elements of the block when LU'd as
well.....
    theFortranBlocks[blockrow][blockcol]-
>InvertMatrix(tempNRCLUmatrix,mylittlevector,

blockwise_diagnolElementsLocal[blockrow]);
  }



  if (blockrow>blockcol) //we are below the diagnol, so we need to
create myBigWorker[blockcol]
  {

    if (maxrow_needed!=0)
      {
      // do:
      //myBigWorker[blockcol]=   theFortranBlock[blockrow][blockcol]    *
theFotranBlock[blockcol][blockcol]
      kblockheight=theFortranBlocks[blockrow][blockcol]->getheight();
      kwidth=theFortranBlocks[blockrow][blockcol]->getwidth();
      kblockwidth=theFortranBlocks[blockcol][blockcol]->getwidth();

      ASSERT(kwidth<=BlockSizes[blockcol]);
      ASSERT(kblockwidth==BlockSizes[blockcol]);

      //first init myBigWorder[blockcol] as all zero....
      for(localrow=1;localrow<=kblockheight;localrow++)
        for(localcol=1;localcol<=kblockwidth;localcol++)
          myBigWorker[blockcol]->Element(localrow,localcol)=0.0;

      //makes the assignment:
      //                  myBigWorker[blockcol]                  <----
theFortranBlocks[blockrow][blockcol]
      //                                                            *
theFortranBlocks[blockcol][blockcol]
      theFortranBlocks[blockcol][blockcol]->OnRightMultMatrix(1.0,

      theFortranBlocks[blockrow][blockcol],
                                          0.0,
                                          myBigWorker[blockcol]);
      workerheight[blockcol]=theFortranBlocks[blockrow][blockcol]-
>getheight();
      workerwidth[blockcol]=theFortranBlocks[blockcol][blockcol]-
>getwidth();
      }
    else // the block we just made was empty thus myBigWorker[blockcol]
is empty as well
      {
      ASSERT(maxcol_needed==0);

      //these assigments signal to us that myBigWorker[blockcol] is
empty.
      workerheight[blockcol]=0;
      workerwidth[blockcol]=0;
      }
  }


  return maxrow_needed*maxcol_needed;

}
```

A.3 Listing of file FortranMatrix.h

```
//----------------------------------------
//----------------------------------------
//  Copyright (c) 2002 Francis X. Canning
//  all rights reserved
//----------------------------------------
//----------------------------------------

#ifndef __FORTRAN__MATRIX__KDEVIL__H_
#define __FORTRAN__MATRIX__KDEVIL__H_

#include "kev_fortran_routines.h"
#include "FullLU.h"
#include <time.h>
#include <math.h>
#include <fstream.h>
#include <iomanip.h>
#include <iostream.h>



// Fortan style matrix: will be stored in a way so that
// passing it to Fortran routines is easy.
// namely it will be stored in Fortran style format, i.e.
// for 1<=i<=height, 1<=j<=width we have:
// data[ i-1 + (j-1)*width] <--> Element(i,j)
class FortranMatrix
```

```
{
private:
  int height, width;
  f_complex *data;

public:
  FortranMatrix(int h, int w);
  ~FortranMatrix();
  f_complex &Element(int i, int j)
    {
      //ASSERT( i>=1 && i<=height);
      //ASSERT( j>=1 && j<=width);
      return data[ i-1 + (j-1)*height];
    }

  f_complex *FortranPtr(void) { return data; }
  int *FortranLeadDim(void) { return &height; }
  int getheight(void) { return height; }
  int getwidth(void) { return width; }
  size_t  MemoryUsed(void)  {  return  sizeof(f_complex)*width*height  +
sizeof(FortranMatrix); }
  void     InvertMatrix(FullLU_NRC     *workspace,f_complex     *temp_Vec,
f_complex *diagstuff=NULL);
  void ApplyToVector(f_complex *in, f_complex *out);
  void OnRightMultMatrix(f_complex alpha, FortranMatrix *leftOp,
                 f_complex beta,  FortranMatrix *outOp);
  void  TruncatedOnRightMultMatrix(int  kblockheight,  int  kblockwidth,
int kwidth,
                            f_complex alpha, FortranMatrix *leftOp,
                            f_complex beta,  FortranMatrix *outOp);

  //makes the assignment:
  // out <--- alpha * (this) * in + beta*out
  void MatrixVectorBlas(f_complex alpha, f_complex beta, f_complex *in,
f_complex *out);
};


#endif
```

A.4 Listing of file FortranMatrix.c

```
//---------------------------------------
//---------------------------------------
//  Copyright (c) 2002 Francis X. Canning
//  all rights reserved
//---------------------------------------
//---------------------------------------

#include "FortranMatrix.h"
```

```
FortranMatrix::FortranMatrix(int h, int w)
{
  int i, totalsize;

  ASSERT(h!=0);
  ASSERT(w!=0);

  height=h;
  width=w;
  totalsize=h*w;

  data=new f_complex[totalsize];
  for(i=0;i<totalsize;i++)
    data[i]=0;
}

FortranMatrix::~FortranMatrix()
{
  delete data;
}

void    FortranMatrix::InvertMatrix(FullLU_NRC    *worker,    f_complex
*temp_Vec, f_complex *diagstuff)
{
  int col,row;

  ASSERT(width==height);
  worker->ChangeDim(height);
  //dump data into worker....
  for(row=1;row<=height;row++)
    for(col=1;col<=width;col++)
      {
      worker->SetElement(row,col,Element(row,col));
      }

  //do LU to worker....
  worker->LUFactor();

  //extract diagnol elements if diagstuff!=NULL
  if(diagstuff!=NULL)
    for (row=1;row<=height;row++)
      diagstuff[row]=worker->GetElement(row,row);

  //first make temp_vec all 0
  for(row=1;row<=height;row++)
    temp_Vec[row]=0;

  for(col=1;col<=width;col++)
    {
      temp_Vec[col]=1;
      // now temp_Vec[i]=0 if i!=col and temp_Vec[col]=1....
      worker->LUSolve(temp_Vec);

      //now temp_Vec is the col'th column of the inverse
```

```
        //copy it into our col'th column and reset temp_Vec as all zeroes
        for(row=1;row<=height;row++)
        {
          Element(row,col)=temp_Vec[row];
          temp_Vec[row]=0;
        }
      }




}

// does the assignment:
// outOp <--- alpha * leftOp * thismatrix  +   beta * outOp
void  FortranMatrix::OnRightMultMatrix(f_complex  alpha,  FortranMatrix
*leftOp,
                           f_complex beta,  FortranMatrix *outOp)
{
  int kblockwidth, kblockheight, kwidth;
  int i,j,k;
  f_complex temp;

  kblockheight=leftOp->getheight();
  kblockwidth=getwidth();
  kwidth=leftOp->getwidth();

  //do it myself, no BLAS
  //for(i=1;i<=kblockheight;i++)
  //  for(j=1;j<=kblockwidth;j++)
  //  {
  //temp=0.0;
  //for(k=1;k<=kwidth;k++)
  //  temp+=leftOp->Element(i,k) * Element(k,j);
  //
  //temp*=alpha;
  //temp+= beta * outOp->Element(i,j);
  //outOp->Element(i,j)= temp;
  //}




   BLAS3_MultIncrMatrix("N","N", //no transpose
         &kblockheight, &kblockwidth, &kwidth,
         &alpha, leftOp->FortranPtr(), leftOp->FortranLeadDim(),
         FortranPtr(), FortranLeadDim(),
         &beta, outOp->FortranPtr(), outOp->FortranLeadDim());


}


void FortranMatrix::ApplyToVector(f_complex *in, f_complex *out)
```

```
{
  int row, incr_one, k;
  f_complex zero, one;

  //do it myself, no BLAS
  //for(row=1;row<=height;row++)
  //  {
  //  out[row]=0.0;
  //  for(k=1;k<=width;k++)
  //out[row]+= Element(row,k) * in[k];
  //  }



  zero=0.0;
  one=1.0;
  incr_one=1;
  for(row=1;row<=height;row++)
    out[row]=0.0;
  BLAS2_MultiplyByMatrix("N", &height, &width, &one,
              FortranPtr(), FortranLeadDim(),
              in+1, &incr_one,
              &zero, out+1, &incr_one);

}



// does the assignment:
// outOp <--- alpha * leftOp * thismatrix  +   beta * outOp
// here outOp is viewed as a matrix with
// height=kblockheight, width=kblockwidth
//  and  (this)  is  viewed  as  having  height=kwidth  and  having
width=kblockwidth
// and leftOp is viewed as having height=kblockheight and width=kwidth
void  FortranMatrix::TruncatedOnRightMultMatrix(int  kblockheight,  int
kblockwidth,
                                           int kwidth,
                                              f_complex  alpha,  FortranMatrix
*leftOp,
                                              f_complex  beta,   FortranMatrix
*outOp)
{
  //int i,j,k;
  //f_complex temp;
  //do it myself, no BLAS
  //for(i=1;i<=kblockheight;i++)
  //  for(j=1;j<=kblockwidth;j++)
  //  {
  //temp=0.0;
  //for(k=1;k<=kwidth;k++)
  //  temp+=leftOp->Element(i,k) * Element(k,j);

  //temp*=alpha;
  //temp+= beta*outOp->Element(i,j);
  //outOp->Element(i,j)=temp;
  //  }




   BLAS3_MultIncrMatrix("N","N", //no transpose
         &kblockheight, &kblockwidth, &kwidth,
         &alpha, leftOp->FortranPtr(), leftOp->FortranLeadDim(),
         FortranPtr(), FortranLeadDim(),
         &beta, outOp->FortranPtr(), outOp->FortranLeadDim());



}


//makes the assignment:
// out <--- alpha * (this) * in + beta*out
void  FortranMatrix::MatrixVectorBlas(f_complex  alpha,  f_complex  beta,
f_complex *in, f_complex *out)
{
  //int row,col,k;
  //f_complex temp;
  //for(row=1;row<=height;row++)
  //{
  //  temp=0.0;
  //  for(k=1;k<=width;k++)
  //temp+= Element(row,k)  *  in[k];
  //  temp*=alpha;
  //  out[row]= beta*out[row]  + temp;
  //}
  //return;


  int incr_one;

  incr_one=1;
  BLAS2_MultiplyByMatrix("N",
                   &height, &width,
                   &alpha, FortranPtr(),
                   FortranLeadDim(),
                   in+1, &incr_one,
                   &beta, out+1, &incr_one);

}
```

A.5 Listing of file kev_fortran_routines.h

```
//--------------------------------------
```

```
//---------------------------------------
//  Copyright (c) 2002 Francis X. Canning
//  all rights reserved
//---------------------------------------
//---------------------------------------

#ifndef __KEV__FORTRAN__ROUTINES__H__
#define __KEV__FORTRAN__ROUTINES__H__

#include "kev_complex.h"
// routines taken from linpack and from BLAS
#ifdef KDEVIL_USE_DOUBLES

//double routine
#define SVDRoutine zsvdc_
extern "C" void zsvdc_(f_complex *x, int *ldx,int *n, int *p,
                    f_complex *s, f_complex *e,
                    f_complex *u, int *ldu,
                    f_complex *v, int *ldv,
                    f_complex *work, int *job, int *info);

extern "C" void zgemv_(char *trans, int *m, int *n,
                     f_complex *alpha, f_complex *Matrix, int *lda,
                     f_complex *x, int *incrx,
                     f_complex *beta, f_complex *y, int *incry);

extern "C" f_complex zdotu_(int *N, f_complex *x, int *incrX, f_complex
*y, int *incrY);
extern "C" f_complex zdotc_(int *N, f_complex *x, int *incrX, f_complex
*y, int *incrY);

extern "C" void zgemm_(char *transposeA, char *transposeB,
                   int *dimM, int *dimN, int *dimK,
                   f_complex *alpha, f_complex *matrixA, int *leadA,
                   f_complex *matrixB, int *leadB, f_complex *beta,
                   f_complex *matrixC, int *leadC);

#define DOTRoutineNotConj zdotu_
#define DOTRoutineConj zdotc_
#define BLAS2_MultiplyByMatrix zgemv_
#define BLAS3_MultIncrMatrix zgemm_
#else

//float routine
extern "C" void csvdc_(f_complex *x, int *ldx,int *n, int *p,
                    f_complex *s, f_complex *e,
                    f_complex *u, int *ldu,
                    f_complex *v, int *ldv,
                    f_complex *work, int *job, int *info);

extern "C" void cgemv_(char *trans, int *m, int *n,
                     f_complex *alpha, f_complex *Matrix, int *lda,
                     f_complex *x, int *incrx,
                     f_complex *beta, f_complex *y, int *incry);
```

```
extern "C" f_complex cdotu_(int *N, f_complex *x, int *incrX, f_complex
*y, int *incrY);
extern "C" f_complex cdotc_(int *N, f_complex *x, int *incrX, f_complex
*y, int *incrY);

extern "C" void cgemm_(char *transposeA, char *transposeB,
                   int *dimM, int *dimN, int *dimK,
                   f_complex *alpha, f_complex *matrixA, int *leadA,
                   f_complex *matrixB, int *leadB,
                   f_complex *beta, f_complex *matrixC, int *leadC);


#define SVDRoutine csvdc_
#define DOTRoutine cdotu_
#define DOTRoutineConj cdotc_
#define BLAS2_MultiplyByMatrix cgemv_
#define BLAS3_MultIncrMatrix cgemm_
#endif

// DOTRoutine is from BLAS, using the FORTRAN interface
// DOTRoutine( int N, f_complex *XArray, int *incX, f_complex *YArray,
int *incY, f_complex *result)
// DOTRoutine returns result as the sum from i=0 to N-1, XArray[
i*incX] * YArray[ i*incY]



#endif
```

A.6 Listing of file kdevil.blockwiseonly.c

```
//---------------------------------------
//---------------------------------------
// Copyright (c) 2002 Francis X. Canning
// all rights reserved
//---------------------------------------
//---------------------------------------


#include <math.h>
#include <time.h>
#include <stdlib.h>
```

```
#include <stdio.h>

// SparseMatrix method:
#include "BlockwiseFactor.h"




//-----------------------------------------
//-----------------------------------------
//  Copyright (c)   Francis X. Canning
//-----------------------------------------
//-----------------------------------------
//   This is an unpublished work containing  confidential and
//   proprietary
//   information of Francis X. Canning and of
//   Simply Sparse Technologies.
//   Disclosure, use, or reproduction without authorization of
//   Francis X. Canning is prohibited.
//-----------------------------------------
//-----------------------------------------
//
//    The C++ code was programmed by Kevin Rogovin
//          krogovin@hotmail.com
//          krogovin@flashcom.com
//          or, c/o Math Dept., Univ of Michigan, Ann Arbor
//-----------------------------------------
//-----------------------------------------

// what is called from fortran (this code supplies)
extern  "C"  void  kdevilf_(int  *matrixDim,  k_real  *wavelegnth,  int
*MethodLeft, int *MethodRight, k_real *tolerance);
extern "C" void kdevils_(f_complex*); // solver called from fortran
extern "C" void inits_(void); // called to initialize the solver
extern  "C"  void  deinit_(void);//  call  to  de-initialize  the  solver  and
factorer, mostly to free memory


// what we need the fortran code to supply:
extern "C" void kdevblk_(int *IB,int *Nblocks); //gives block structure
extern "C" void getmatrix_(f_complex *data, int *leadz,
                           int *numrows, int *whichrows,
                           int *numcols, int *whichcols);

extern "C" void kdevren_(int *); // gives reordeing info


//  routines  provided  by  Fortran  to  generate  change  of  co-ordinate
matrix elements:
extern    "C"    void    kdevfld_(k_real    *Theta,    k_real    *Phi,int
*InonZ,f_complex *ETH, f_complex *EPH);
extern    "C"    void    kdevtst_(k_real    *Theta,    k_real    *Phi,int
*InonZ,f_complex *ETH, f_complex *EPH);

// "names" of above routines
```

```
#define KDEVIL_USE_FIELD 0
#define KDEVIL_USE_COORDINATES 1

// the list of routines, and thier names that generate the change of
co-ordiante matrices.
MethodFunction FunctionList[]= { kdevfld_, kdevtst_ };
typedef char string[30];
string FunctionNames[]= { "kdevfld_", "kdevtst_ " };
int MaxNumName=1;


static KDevilBase *FMatrix; // the actual matrix...



// here we are going to create a front end for fortran!
class FInterface:virtual public KDevilBase
{
public:
      FInterface() {}
      virtual ~FInterface() {}
      void getmatrix_(f_complex *data, int *leadz,
                              int *numrows, int *whichrows,
                              int *numcols, int *whichcols)
      {   ::getmatrix_(data,   leadz,   numrows,   whichrows,   numcols,
whichcols); }

#ifdef KDEVIL_USE_WEIGHTS_ //only used if weights are allowed...
      // only provide this routine if compiling with weight support,
      // code it as if it was a Fortran routine...
      void kdevwts_(k_real *arg1, k_real *arg2)
      {
            ::kdevwts_(arg1,arg2);
      }
#endif

};


class  BlockwiseFInterface:virtual  public  FInterface,  virtual  public
BlockwiseFactor
{
public:
  BlockwiseFInterface(char *initFile, char *diagfile=NULL);
  virtual ~BlockwiseFInterface() {}
};


BlockwiseFInterface::BlockwiseFInterface(char             *initFile,char
*diagfile):FInterface(),
BlockwiseFactor(initFile, diagfile) {}
```

```
extern "C" void kdevilf_(int *sz, k_real *wl, int *MLeft, int *MRight,
k_real *tolerance)
{
      ifstream inKdevilfile;

      int p_N, p_nblocks;
      k_real p_tol, p_wl, p_tolLeft, p_tolRight;
      int *temphuge, *p_UndoReorderInfo;
      int p_numLeft, p_numRight;
      MethodFunction p_LeftMethod, p_RightMethod;
      int sparseMethod;


      cout << "-----------------------------------------\n";
      cout << "-----------------------------------------\n";
      cout << "  Copyright (c)   Francis X. Canning\n";
      cout << "-----------------------------------------\n";
      cout << "-----------------------------------------\n";
      cout << "   This is an unpublished work containing  confidential
and\n";
      cout << "   proprietary\n";
      cout << "   information of Francis X. Canning and of
\n";   cout << "   Simply Sparse Technologies.\n";
      cout   <<   "       Disclosure,   use,   or   reproduction   without
authorization of\n";
      cout << "   Francis X. Canning is prohibited.\n";
      cout << "-----------------------------------------\n";
      cout << "-----------------------------------------\n";




      p_N=*sz;
      p_wl=*wl;



      p_UndoReorderInfo=new int[p_N];
      kdevren_(p_UndoReorderInfo);
      p_UndoReorderInfo--;

      ///////////////// get block info
      cout << "\nGetting Block Info.."; cout.flush();
      temphuge=new int[p_N];
      kdevblk_(temphuge,&p_nblocks);
      temphuge--;
      cout  <<  "Completed,  there  are  "  <<  p_nblocks  <<  "  blocks";
cout.flush();


      inKdevilfile.open("kdevil.ini");
      if (inKdevilfile)
      {
            cout  <<  "\nFound  kdevil.ini  file,  reading  in  tolLeft,
tolRight, numLeft, numRight from file.\n";
            cout.flush();
            inKdevilfile >> p_tolLeft;
            inKdevilfile >> p_tolRight;
            inKdevilfile >> p_numLeft;
            inKdevilfile >> p_numRight;
            inKdevilfile >> p_tol;

            if (inKdevilfile)
                  inKdevilfile >> sparseMethod;
            else
                  sparseMethod=-1; // use default method, CustomSparse,
                                                      //      don't      output
DiagnolElements.


            inKdevilfile.close();

      }
      else
      {
            cout  <<  "\nCould  not  open  file  kdevil.ini  for  reading,
using defaults\n";
            p_tolLeft  = *tolerance;
            p_tolRight = *tolerance;
            p_numLeft  = 5;
            p_numRight = 5;
            sparseMethod=-1;
            p_tol=*tolerance;
      }



      ASSERT(*MLeft<=MaxNumName);
      cout << "\nLeftMethod =" << *MLeft   << " \" " << FunctionNames
[*MLeft] << " \" ";
      cout << "\nnumLeft =" << p_numLeft << " tolLeft=" << p_tolLeft;

      ASSERT(*MRight<=MaxNumName);
      cout  <<  "\nRightMethod="  <<  *MRight   <<   "   \"   "   <<
FunctionNames[*MRight] << " \" ";
      cout  <<  "\nnumRight  ="  <<  p_numRight  <<  "  tolRight="  <<
p_tolRight;

      p_LeftMethod=FunctionList[*MLeft];
      p_RightMethod=FunctionList[*MRight];




      cout <<"\n Using Blockwise packaage------\n";
      FMatrix=new                    BlockwiseFInterface("Blockwise.ini",
"NumericalDiagnolElements");
      cout.flush();
      FMatrix->Init(p_N, p_nblocks, temphuge, p_tol, p_wl,
                  p_LeftMethod, p_RightMethod,
                  p_numLeft , p_tolLeft,
                  p_numRight, p_tolRight,
                  p_UndoReorderInfo);



      //////////////////////////////////////////
      // to change what is outputted where, one should
      // insert the code here, for example,
      // FMatrix->OutputLeftSingularValues="MyFavoriteFileName";
      // would make the LeftSingular values to be outputted
      // to the file MyFavoriteFileName, while,
      // FMatrix->OutputLeftSingularValues=NULL;
      // would be make it so that the LeftSingular values are
      //  not  written  to  any  file.  there  are  6  files  that  one  can
assign, they are:
      //
      //  FMatrix->OutputLeftSingularValues      ===>  for  left  singular
values
      // FMatrix->OutputRightSingularValues     ===> for right  singular
values
      // FMatrix->OutputBlockData          ===>  for  BlockData,  eg  how
far each block was calculated,
      //                                          ala tolLeft,numLeft,
tolRight, numRight
      // FMatrix->OutputTimingData              ===>   for   the   timing
data, how long differenf actions took
      // FMatrix->OutputHistogram                     ===>    for    the
histogra
      // FMatrix->OutputMatrix                        ===>    for    the
sparse matrix BEFORE LU factorization.
      //
      // if one makes no assignments, then the default behavior is to
output,
      // the filename is Numerical???? where ????
      //  corrosponds  to  one  of  LeftSigularValues,  RightSigularValues,
BlockData
      // TimingData, HistoGramData, UnfactoredMatrix


      FMatrix->CreateFactoredSparseMatrix();


      // init the solver as well,
      inits_();

      cout  <<"\nLeaving kdevil_"; cout.flush();
```

```
}


void kdevils_(f_complex *rhs)
{
  //rhs is from fortran, so we need to decrement the pointer
  rhs--;
  FMatrix->Solver(rhs);


}

void deinit_(void)
{
  cout << "\nDoing DeInit..."; cout.flush();
  FMatrix->Deinit();
  delete FMatrix;
  cout << "Done."; cout.flush();
}

void inits_(void)
{
      static int inits_called=0;

      if (inits_called)
            return;


      inits_called=1;
      cout << "\nIniting solver.."; cout.flush();
      FMatrix->InitSolver();
      cout << "Done."; cout.flush();
}
```